

conference

proceedings

**The Twelfth Systems
Administration Conference
(LISA '98) Proceedings**

*Boston, Massachusetts
December 6-11, 1998*

Co-sponsored by **The USENIX Association** and
SAGE, the System Administrators Guild

USENIX[®]

The Advanced Computing
Systems Association

SAGE
THE SYSTEM ADMINISTRATORS GUILD

USENIX

Twelfth Systems Administration Conference (LISA '98) Proceedings

Boston, Massachusetts

December 1998

For additional copies of these proceedings contact

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Telephone: 510-528-8649

The price is \$32 for members and \$40 for nonmembers.

Past USENIX Large Installation Systems Administration Workshop
and Conference Proceedings (price: member/nonmember)

Large Installation Systems Admin. I Workshop	1987	Philadelphia, PA	\$4/\$4
Large Installation Systems Admin. II Workshop	1988	Monterey, CA	\$8/\$8
Large Installation Systems Admin. III Workshop	1989	Austin, TX	\$13/\$13
Large Installation Systems Admin. IV Conference	1990	Colorado Spgs, CO	\$15/\$18
Large Installation Systems Admin. V Conference	1991	San Diego, CA	\$20/\$23
Systems Administration VI Conference	1992	Long Beach, CA	\$23/\$30
Systems Administration VII Conference	1993	Monterey, CA	\$25/\$33
Systems Administration VIII Conference	1994	San Diego, CA	\$22/\$29
Systems Administration IX Conference	1995	Monterey, CA	\$30/\$38
Systems Administration X Conference	1996	Chicago, IL	\$30/\$38
Systems Administration XI Conference	1997	San Diego, CA	\$30/\$38

Outside the U.S.A. and Canada, please add \$12
per copy for postage (via air printed matter).

Copyright © 1998 by The USENIX Association. All rights reserved.

This volume is published as a collective work.

Rights to individual papers remain with the author or the author's employer.

Permission is granted for the noncommercial reproduction of the
complete work for educational or research purposes.

ISBN 1-880446-40-5

AIX is a trademark of IBM.

Adaptive Server is a trademark of Sybase, Inc.

Backup Server is a trademark of Sybase, Inc.

Cisco is a trademark of Cisco Systems, Inc.

Enterprise is a trademark of Sun Microsystems, Inc.

IBM RS/6000 is a trademark of IBM.

Java is a trademark of Sun Microsystems.

ORACLE PL/SQL is a trademark of Oracle, Inc.

Oracle is a trademark of Oracle, Inc.

Oracle8 is a trademark of Oracle, Inc.

PostScript is a trademark of Adobe Corporation.

SGI is a trademark of Silicon Graphics, Inc.

SQL Server is a trademark of Sybase, Inc.

SQL*Net is a trademark of Oracle Corporation.

Sniffer is a trademark of Network General.

Solaris is a trademark of Sun Microsystems, Inc.

Sun is a trademark of Sun Microsystems, Inc.

Sybase is a trademark of Sybase, Inc.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Co. Ltd.

VERITAS File System is a trademark of VERITAS Software Corporation.

VERITAS FirstWatch is a registered trademark of VERITAS Software Corporation.

VERITAS Volume Manager is a trademark of VERITAS Software Corporation.

VERITAS is a registered trademarks of VERITAS Software Corporation.

USENIX acknowledges all trademarks appearing herein.

 Printed in the United States of America on 50% recycled paper, 10-15% post consumer waste.

USENIX Association

**Proceedings of the Twelfth
Systems Administration Conference
(LISA XII)**

**December 6-11, 1998
Boston, MA, USA**

TABLE OF CONTENTS

Acknowledgments	iv
Preface	v
Author Index	vi

Opening Remarks

Wednesday (9:00-10:30 am) Chairs: Xev Gittler and Rob Kolstad

Security

Wednesday (11:00am-12:30pm) Chair: Phil Cox

TITAN	1
<i>Dan Farmer, Earthlink Network; Brad Powell, Sun Microsystems, Inc.; Matthew Archibald, KLA-Tencor</i>	
Infrastructure: A Prerequisite for Effective Security	11
<i>Bill Fithen, Steve Kalinowski, Jeff Carpenter, and Jed Pickel, CERT Coordination Center</i>	
SSU: Extending SSH for Secure Root Administration	27
<i>Christopher Thorpe, Yahoo!, Inc.</i>	

Pushing Users and Scripts Around

Wednesday (2:00pm-3:30pm) Chair: Ozan S. Yigit

System Management With NetScript	37
<i>Apratim Purakayastha and Ajay Mohindra, IBM T. J. Watson Research Center</i>	
Accountworks: Users Create Accounts on SQL, Notes, NT, and UNIX	49
<i>Bob Arnold, Sybase, Inc.</i>	
Single Sign-On and the System Administrator	63
<i>Michael Fleming Grubb and Rob Carter, Duke University</i>	

Storage Performance

Wednesday (4:00pm-5:30pm) Chair: Marc Staveley

Using Gigabit Ethernet to Backup Six Terabytes	87
<i>W. Curtis Preston, Hughes Space and Communications</i>	
Configuring Database Systems	97
<i>Christopher R. Page, Millennium Pharmaceuticals</i>	

Distributed Computing

Thursday (9:00am-10:30am)

Chair: Phil Cox

A Configuration Distribution System for Heterogeneous Networks	109
<i>Glêdson Elias da Silveira, Federal University of Rio Grande do Norte; Fabio Q. B. da Silva, Federal University of Pernambuco</i>	
An NFS Configuration Management System and its Underlying Object-Oriented Model	121
<i>Fabio Q. B. da Silva, Juliana Silva da Cunha, Danielle M. Franklin, Luciana S. Varejão, and Rosalie Belian, Federal University of Pernambuco</i>	
Design and Implementation of an Administration System for Distributed Web Server	131
<i>C. S. Yang and M. Y. Luo, National Sun Yat-Sen University, Taiwan, R.O.C.</i>	

Networking

Thursday (11:00am-12:30pm)

Chair: Eric Anderson

MRTG – The Multi Router Traffic Grapher	141
<i>Tobias Oetiker, Swiss Federal Institute of Technology, Zurich</i>	
Wide Area Network Ecology	149
<i>Jon T. Meek, Edwin S. Eichert, Kim Takayama, Cyanamid Agricultural Research Center/American Home Products Corporation</i>	
Automatically Selecting a Close Mirror Based on Network Topology	159
<i>Giray Pultar, giray@coubros.com</i>	

Infrastructure

Thursday (2:00pm-3:30pm)

Chair: John Orthoefer

What to Do When the Lease Expires: A Moving Experience	167
<i>Lloyd Cha, Chris Motta, Syed Babar, and Mukul Agarwal, Advanced Micro Devices, Inc.; Jack Ma and Waseem Shaikh, Taos Mountain, Inc.; Istvan Marko, Volt Services Group</i>	
Anatomy of an Athena Workstation	175
<i>Thomas Bushnell, BSG and Karl Ramm, MIT Information Systems</i>	
Bootstrapping an Infrastructure	181
<i>Steve Traugott, Sterling Software and NASA Ames Research Center; Joel Huddleston, Level 3 Communications</i>	

Printing and Configuring Files

Thursday (4:00pm-5:30pm)

Chair: David Kensiski

Ganymede: An Extensible and Customizable Directory Management Framework	197
<i>Jonathan Abbey and Michael Mulvaney, The University of Texas at Austin</i>	
Building An Enterprise Printing System	219
<i>Ben Woodard, Cisco Systems</i>	
Large Scale Print Spool Service	229
<i>Ignacio Reguero, David Foster, and Ivan Deloose, CERN</i>	

Distributing Software Packages

Friday (9:00am-11:00am)

Chair: E. Scott Menter

mkpkg: A software packaging tool	243
<i>Carl Staelin, Hewlett-Packard Laboratories</i>	
SEPP – Software Installation and Sharing System	253
<i>Tobias Oetiker, Swiss Federal Institute of Technology, Zurich</i>	
Synctree for Single Point Installation, Upgrades, and OS Patches	261
<i>John Lockard, University of Michigan; Jason Larke, ANS Communications, Inc.</i>	

New Thoughts and Evolution

Friday (11:00am-12:30 pm)

Chair: Melissa D. Binde

The Evolution of the CMD Computing Environment: A Case Study in Rapid Growth	271
<i>Lloyd Cha, Chris Motta, Syed Babar, and Mukul Agarwal, Advanced Micro Devices, Inc.; Jack Ma and Waseem Shaikh, Taos Mountain, Inc.; Istvan Marko, Volt Services Group</i>	
Computer Immunology	283
<i>Mark Burgess, Oslo College</i>	
A Visual Approach for Monitoring Logs	299
<i>Luc Girardin and Dominique Brodbeck, UBS, Ubilab</i>	

Mailing Lists

Friday (2:00pm-3:30pm)

Chair: Tim Hunter

Mailman: The GNU Mailing List Manager	309
<i>John Viega, Reliable Software Technologies; Barry Warsaw and Ken Manheimer, Corporation for National Research Initiatives</i>	
Drinking from the Fire(walls) Hose: Another Approach to Very Large Mailing Lists	317
<i>Strata Rose Chalup, Christine Hogan, Greg Kulosa, Bryan McDonald, and Bryan Stansell, Global Networking and Computing, Inc.</i>	
Request v3: A Modular, Extensible Task Tracking Tool	327
<i>Joe Rhett, Navigist</i>	

ACKNOWLEDGMENTS

PROGRAM CO-CHAIRS

Xev Gittler, *Goldman, Sachs & Co.*
Rob Kolstad, *The SANS Institute*

PROGRAM COMMITTEE

Eric Anderson, *Univ. of California, Berkeley*
Melissa D. Binde, *Amazon.com*
Phil Cox, *NTS, Inc.*
Tim Hunter, *KLA-Tencor Corp.*
David Kensiski, *Digital Island, Inc.*
Kurt J. Lidl, *UUNET Technologies, Inc.*
E. Scott Menter, *ESM Services, Inc.*
John Orthoefer, *GTE Internetworking*
John Sellens, *UUNET Canada, Inc.*
Marc Staveley, *Sun Microsystems, Inc.*
Ozan S. Yigit, *Sun Microsystems, Inc.*

READERS

Strata Rose Chalup, *VirtualNet Consulting*
Rik Farrow, *Independent Consultant*
David Nochlin, *UBS, Inc.*
David Parter, *University of Wisconsin*
Gretchen Phillips, *University of Buffalo*

INVITED TALKS COORDINATORS

Phil Scarr, *Global Networking and Computing*
Pat Wilson, *Dartmouth College*

WORK-IN-PROGRESS COORDINATOR

Peg Schafer, *Harvard University*

GLOBAL LISA

Joel Avery, *Nortel, Inc.*
Rob Kolstad, *The SANS Institute*

GURU-IS-IN COORDINATOR

Lee Damon, *QUALCOMM, Inc.*

ADVANCED TOPICS

Adam Moskowitz, *Genome Therapeutics Corp.*

PRACTICUM COMMITTEE

Lee Damon, *QUALCOMM, Inc.*
David Parter, *University of Wisconsin*
Strata Rose Chalup, *VirtualNet Consulting*
Rob Kolstad, *The SANS Institute*
Gretchen Phillips, *University of Buffalo*
Adam Moskowitz, *Genome Therapeutics Corp.*

TERMINAL ROOM COORDINATOR

Lynda McGinley, *University of Chicago*

PROCEEDINGS PRODUCTION

Rob Kolstad, *The SANS Institute*
Data Reproductions

USENIX ASSOCIATION

Ellie Young, *Executive Director*
Judith F. DesHarnais, *Conference Coordinator*
Daniel V. Klein, *Tutorial Coordinator*
Cynthia Deno, *Marketing and Exhibitions*

USENIX SUPPORT STAFF

Linda Barnett, *USENIX Association*
Cami Edwards, *USENIX Association*
Lana Erlanson, *USENIX Association*
Vanessa Fonseca, *USENIX Association*
Toni Veglia, *USENIX Association*

PREFACE

Welcome to Boston! We've planned the biggest and best LISA ever. We've assembled a program that includes almost 30 of the best papers, the now-traditional invited talks track, and the new Practicum track with outstanding speakers and timely topics.

Special thanks to Judy DesHarnais, the USENIX conference coordinator, who has organized the entire week of facilities and amenities. Special thanks also to Dan Klein, the USENIX tutorial coordinator, who has again assembled a super set of courses with terrific instructors.

The Conference Proceedings, lovingly chosen by the Program Committee and submitted by the various speakers, dwarfs its predecessors in size. Thanks to all who participated: those who submitted abstracts, those who reviewed them, and those who wrote final papers and created presentations.

The Invited Talks were marshaled this year by Phil Scarr and Pat Wilson who threaded their way through the minefield of conflicting topics and schedules to create a great track.

The new Practicum committee invited a slightly different style of speakers in the very pragmatic mode for a third track. We can't help but believe that this track will succeed stupendously.

Of course, this conference would be happening under a tent in a field without the incredible efforts of the USENIX Office. We probably wouldn't even have the tent if it were not for the efforts of folks from both the Conference and Administrative offices. Thanks to them!

Thanks for coming to Boston! See you in the hallways!

Xev Gittler

Rob Kolstad

AUTHOR INDEX

Jonathan Abbey	197	Jack Ma	167
Mukul Agarwal	167	Jack Ma	271
Mukul Agarwal	271	Ken Manheimer	309
Matthew Archibald	1	Istvan Marko	167
Bob Arnold	49	Istvan Marko	271
BSG	175	Bryan McDonald	317
Syed Babar	167	Jon T. Meek	149
Syed Babar	271	Ajay Mohindra	37
Rosalie Belian	121	Chris Motta	167
Dominique Brodbeck	299	Chris Motta	271
Mark Burgess	283	Michael Mulvaney	197
Thomas Bushnell	175	Tobias Oetiker	141
Jeff Carpenter	11	Tobias Oetiker	253
Rob Carter	63	Christopher R. Page	97
Lloyd Cha	167	Jed Pickel	11
Lloyd Cha	271	Brad Powell	1
Strata Rose Chalup	317	W. Curtis Preston	87
Juliana Silva da Cunha	121	Giray Pultar	159
Fabio Q. B. da Silva	109	Apratim Purakayastha	37
Fabio Q. B. da Silva	121	Karl Ramm	175
Ivan Deloose	229	Ignacio Reguero	229
Edwin S. Eichert	149	Joe Rhett	327
Dan Farmer	1	Waseem Shaikh	167
Bill Fithen	11	Waseem Shaikh	271
David Foster	229	Glêdson Elias da Silveira	109
Danielle M. Franklin	121	Carl Staelin	243
Luc Girardin	299	Bryan Stansell	317
Michael Fleming Grubb	63	Kim Takayama	149
Christine Hogan	317	Christopher Thorpe	27
Joel Huddleston	181	Steve Traugott	181
Steve Kalinowski	11	Luciana S. Varejão	121
Greg Kulosa	317	John Viega	309
Jason Larke	261	Barry Warsaw	309
John Lockard	261	Ben Woodard	219
M. Y. Luo	131	C. S. Yang	131

TITAN

*Dan Farmer – Earthlink Network
Brad Powell – Sun Microsystems, Inc.
Matthew Archibald – KLA-Tencor*

ABSTRACT

Titan is a freely available host-based security tool that can be used to improve or audit the security of a UNIX system. It was written almost completely in Bourne shell, with a master script controlling the execution of many smaller programs. Each of the programs either fixes or detects potential security problem, and its simple and extremely modular design also makes it useful to help check or enforce the adherence of a system against its security policy. Finally, anyone who can write a shell script or program can easily create their own Titan modules.

Titan does not replace other security tools, nor does it fix or patch security bugs; its primary purpose is to improve the security of the system it runs on by codifying as many security tricks to secure an OS that the authors could think of. And when used in combination with other security tools it can help make the transformation of an "out of the box" system into a firewall or security conscious system a significantly easier task.

NOTE: Due to time, resource, and expertise limitations, the first release of Titan is only known to run on Solaris Operating Systems, versions Solaris 2.x and Solaris 1.x. However, many of the small sub-programs within Titan work well with other UNIX's, and other than taking the time to create Titan modules for them, there is nothing Sun specific about Titan that would prevent it working on other UNIX systems.

Introduction

UNIX is often, and justifiably, criticized for being a difficult system to administer because it is not only complex and cantankerous but hard to secure. Its enormous configurability, the fact that vendors don't ship secure systems, and that it requires significant amounts of time, resources, and expertise to safeguard a host are only some of the reasons that so many UNIX systems are insecure on the Internet. To compound the problem, like all modern operating systems it not only becomes less secure as time goes on (simply due to usage), but with the rapidly changing security field, it also requires considerable effort to stay abreast of the latest information – time that most system administrators simply don't have.

Titan tries to provide at least a partial solution to all these problems by trying to locate and fix many of the more common procedural problems that crop up, as well as put into one place all those damn OS tweaks that can assist in securing your system. Titan improves the security of a system by:

- Cutting off entry points into the system.
- Mitigating or preventing the effects of various denial of service (DOS) attacks.
- Turning on or improving the level of logging and auditing features.
- Improving network and local (e.g., host level) defenses.
- Assisting in programmatically defining and enforcing a system security policy.

It is important to note that Titan's focus is the correction of procedural problems. While it can be used as an adjunct to other auditing tools, whether host or network based, it does not attempt to find problems that it cannot correct. An automated tool that changed weak passwords, unpatched or insecure system binaries, and unrestricted filesystem mounts, for instance, could break or disrupt operations to an unacceptable level. Like most other security tools, Titan is not meant to be used only once: to achieve effective security requires an ongoing concern and continued attention to good security practices. Any competent system administrator should have considered, if not resolved or repaired, nearly all of the problems that Titan addresses on their security critical systems.

Anyone working in security or systems administration who has been around the Internet for any length of time has done it – making the same changes, over and over again, to secure a system. Worse yet, each new OS release brings tiny, seemingly arbitrary changes that can invalidate prior work. And forget it when a major new release comes out, or you have to work with another operating system altogether! Just among the authors of Titan we've ftp'd Crack, COPS, and other security programs from the net thousands of times – and we're sure we're not alone.

The analysis of the security of a system is depressing – the same sets of problems always come up. But what's worse is that these problems can almost always be easily fixed – so why aren't they? And the final sling of indignity is that vendors keep changing

the damn commands to do the same things, even within the same major version of the OS (what are the arguments to *ndd(1M)* that change that TCP behavior?) And it's certainly not only Sun – it's DEC, it's HP, it's IBM, it's everyone that has even a mildly complex system. Yes, even Microsoft.

So why do these same problems show up over and over, regardless of the supplier? Good question! We don't know the answer, but what we do know is that having a tool to help ensure your system's consistency is a very positive step in the right direction. Hence Titan.

Titan's main design goals are:

- **Security comes first.** We can mandate that for this tool, Security comes first. After Titan has been run on a system it should be more secure than before and there will be no remaining significant host level security problems that we know of, other than those involving vendor OS and independent daemon security issues and patches (e.g., if the system is a WWW server, CGI or CGI-like interfaces could be problematic). The system will not be 100% secure – none are – but it will be pretty darn secure, especially after applying security patches. Due to customer pressure vendors can't take the chance with their patches and system releases that things will break – but we can. In our testing and use over the years we haven't run into a single thing that Titan has irreparably broken, but it certainly could happen.
- **Easy to use.** Titan should be simple to install and run. While knowledge about the system will always help, you can trust Titan to do the right thing in most cases.
- **Policy based.** Titan can assist in the creation of a programmatically defined technical system or site security policy. Classes or types of security (such as firewall, desktop, etc.) are simple to define and apply to the appropriate system, and help produce a consistently secure system in ways that are readily comprehensible.
- **Freely available source code.** In security it is imperative to have complete source code availability. Having full control over what is run and possessing the potential for total understanding of exactly what actions it performs is mandatory for a truly effective security tool.
- **Modular.** Titan is non-monolithic and easily extended. Shell scripts or other programs can either be taken out of or added into Titan's framework. Doing so will not affect the other programs.
- **Useful.** Quite simply, we've found Titan to be enormously handy and something that can be used quite frequently if security is a significant concern for you or your systems.

We're often asked how to tighten down the OS when a firewall product gets installed. There is a

reasonable expectation from the customer that after the firewall is installed, the system will not be compromised by an attack that is outside the scope of the firewall product. After all, aren't firewalls supposed to protect you? You wouldn't say it was safe to run a business on the Internet unless you could protect it, would you? Unfortunately most people don't know what security is, and the firewall sales people are not going to help.

And it really is unreasonable to expect the user, a customer, to understand all, or even most of the security issues of running a system on the Internet. Why should they? However, this does place both the specific firewall vendor and security people in general in a rather awkward situation. Indeed, what probably scares firewall vendors more than anything else is the fact that firewalls are failing because users or administrators don't fix, remove, or upgrade old versions of a potentially vulnerable network service.

Titan Does Not . . .

It is important to note that there are several procedural security problems that Titan does not attempt to fix or seal off. CGI programs, often deployed on WWW servers, are becoming one of the more widespread security problems on the Internet, and are nearly impossible to programmatically detect or prevent. Titan also does not render a system impervious to breakins – not only are inside attacks common, but new bugs and holes are constantly being found. Remember – there is an arms race going on out there.

In addition, Titan does not address the problems of secure software distribution or updates. This means that Titan is probably not a viable tool to secure all systems on a large network due to the administrative costs involved in setting up and maintaining all the copies of Titan. While NFS, *rdist*, or other methods of software replication and deployment could be used, we would warn that the inherent security risks of such methods, as well as the myriad dangers of controlling an organization from a central location probably outweigh the benefits in most situations.

Using Titan

Using Titan is fairly simple, but we want to reiterate – it cannot secure a system, nor can it fix problems that will inevitably arise unless you continue to run it. We suggest that you employ the following sequence when first utilizing Titan:

- Read the Titan documentation and look at the programs. Does it do what you want? Does it fit into your security policy?
- Examine or secure your system using your normal set of tools and procedures. Note any flaws.
- Back up your system.
- Run Titan.
- Examine your system again. Are the flaws

gone? Are new ones there? Does everything still work?

- Install strong authentication on your system, at least for remote logins. Anyone using the same reusable password in cleartext over the Internet except in an emergency is a fool.
- Continually monitor your system, running Titan and other security tools as well as applying security patches as necessary or as your security policy changes over time.
- Report any problems found with Titan to us so we can fix them in subsequent releases!

After the initial use of Titan we suggest running it in the verify mode at least once a week. You can run Titan from cron in the fix mode, but since it can affect your system drastically we would suggest being cautious about doing so (in addition, if you run the Tripwire integrity checking tool, it will complain vigorously about all the files Titan changes). In addition, we highly recommend that stronger authentication (such as with the Logdaemon package or hardware methods) be installed and utilized on the system. When data is traversing the network, strong encryption should be used, if possible, in addition to the extra authentication.

If Titan does the majority of things that you personally do to secure your systems but misses a few points, you might consider writing some additional Titan modules to perform the tasks. We would also ask you to send us email about this – if security related, we would certainly consider including your module in the general Titan release or rewriting it ourselves.

A Case Study

Lucinda Williams is a ten year veteran of system administration and security on the Internet, and has been recently appointed chief security officer for the medical center of her alma mater, Evil University (Evil U, aka evil.edu). After modifying Evil U's general security policy to fit in with the needs of her constituents, she has started to implement the technical aspects of the program. Since the university is strapped for cash, the firewall (an Ultra running Solaris 2.6) must also serve as a WWW server. Here are the steps she takes to create and secure the medical center's firewall:

- A new system is installed with the absolute minimum number of options required to run the system, which lives on its own subnet to prevent local packet sniffing. Immediately, inetd is (temporarily) disabled to help ward off intruders attacking the system before it has been properly secured. As soon as she downloads all the security tools and files needed to install the system, it is physically disconnected from the network. GCC (the GNU C compiler) is also installed so that various security tools can be compiled. (She could alternately compile them on another system that is known to be

uncompromised and ftp the binaries over, but it's safer to compile them locally to help ensure that they have not been tampered with.) The Apache httpd server is installed because of its good security and source code availability. The most recent version of sendmail is then put into place.

- The packet screen is the first defensive tool set up. Almost without exception, any critical system that is not protected by a screening router, proxy firewall, or both (or, less ideally, a program such as screend or IP Filter that duplicates this function) has not been adequately protected. While not sufficient to secure a system or network by itself, it is a necessary part of any security solution. Under most circumstances the router shouldn't have to allow more than a half-dozen ports or so from the outside world. DNS, SMTP, http, telnet, and some ICMP is all Lucinda allows, although she is forced by Evil U's policy to allow NFS, Net-BIOS, finger, and ftp to the rest of the University.
- At the time of this writing, ftp://sunsolve1.sun.com/pub/patches/ contains all the Sun OS and program patches (including many security fixes). Sun also provides a description of and a tar file containing all their recommended patches for their released OS's, which Lucinda ftp's and installs. This is done before running Titan, since the patches might undo some of the system modifications Titan performs.
- She has previously checked Titan against her firewall security policy, and has had to make a few small changes:
 - The issues file needed revision to reflect Evil U's administrative policies.
 - She needs to allow root ftp access (a truly abysmal idea, but one required by university policy), so in the ftpusers.sh Titan module, she removes root from the \$DEFFTPUSERS variable.
 - The university has a customized (and mandatory) compiled C program that all systems must run via the /etc/aliases file for inventory purposes. She has several choices here – she can modify the Titan module (aliases.sh) that doesn't like mail aliases that point to a program, ignore the warnings, or not run the alias module at all. The last can be accomplished either by creating a Titan policy file with all the modules that she does want to run, or by simply moving the shell script out of the modules directory.
- She runs Titan with the -f flag to fix all the problems it detects, and then installs Titan in cron to run with the -v flag once per week.
- Logdaemon – despite being vulnerable to

session hijacking and eavesdropping – is used to improve the authentication of all users instead of the popular but much more complex and potentially dangerous ssh. All accounts have their normal UNIX reusable password disabled.

- Next she runs Tiger and/or COPS, fixes any problems found, and creates a cron job to run the tool once per day, mailing the results to her.
- Logging tools are then set in place next. The TCP and portmapper wrappers and swatch are all installed, with syslog sending information both locally and to a central server, and furthermore any critical events are mailed to Lucinda's pager.
- As the final step in the setup process, she removes GCC and makes a full backup of the system, storing it off-site.

The system is now ready to run. She'll test the initial security with a remote security scanner that is run on the outside of her domain (and hopefully outside the university). Any of the widely available programs such as SATAN, ISS, SAINT, or CyberGuard would work, depending on her familiarity with these and her budget. Initially, the port scanner is the most important thing to run. She also subscribes to the Bug-Traq, Sun security advisory, and CERT mailing lists, and will keep a close eye on the system logs and activity. She has also created an addendum to her local security policy that will require any and all CGI programs to be audited and personally approved by her as well as being placed in sbx (a CGI safety wrapper). If all this is done, the system shouldn't take too much time to set up and continue to run, and should be a very secure system. The rest of Evil U is perhaps her largest security concern, since they have significant access to the rest of the network she maintains, but there is little she can do but use the TCP and other wrappers and auditing tools to watch the traffic.

NFR, tcpdump, or other packet watching tools can be potentially marvelous tools, but do require a significant time investiture to run effectively. The widespread availability of very inexpensive large high speed disks (to save the voluminous audit data) does make the process more viable, however.

Titan Features

Although Titan has been a dynamic system, continually adding features and additional fixes or tests, we feel it important to cover some of the more interesting tests or features. Code fragments will frequently be given, but with any of the problems listed below, looking at the source code of the corresponding Titan sub-program can be illuminating.

The following sections discuss some of the changes that Titan makes to a system. However, any list we could create will be out of date fairly soon – the complete and up to date list can be found at the

online Titan documentation. Since Solaris 2.x contains several ways to improve a host's security that earlier versions of Solaris did not, we naturally have more Titan modules for it – and generally recommend running it or a similar system if security is a concern.

Kernel Level Configuration

Why is it that almost every proxy firewall we see has `ip_forward_src_routed` enabled? Source routing and other such options may have their uses, and at one time were fine ideas, but they do not belong in the world of Internet security, unless you're trying to circumnavigate it. Tools to abuse and bypass systems that aren't sewn up tightly proliferate on the Internet.

Most modern UNIX's allow a great deal of kernel tuning from the command line. Solaris, for instance, has `ndd(1M)`, which can get and set configuration parameters in TCP kernel drivers. Putting them in the `/etc/system` file makes the change take effect at boot time. Titan closes various kernel and TCP/IP protocol holes that we're aware of, including:

- Fixing the stack. Ever since Aleph1's pivotal paper detailing how to exploit buffer overflows, stack smashing programs have perhaps become the most common type of exploited coding error. Solaris allows the kernel stack to be non executable; it adds the following entry into `/etc/system` so that zero-fill-on-demand pages are marked `rw-` instead of `rw-x`:

- Don't allow executing code on the stack

```
set noexec_user_stack = 1
```

- And log it when it happens.

```
set noexec_user_stack_log = 1
```

- NFS bind. Titan sets the privileged port definition to all ports above 2050. NFS, which uses UDP port 2049, has been historically set in an unsafe port range. If you want to protect other services above this range, simply change this parameter.

```
ndd -set /dev/udp \
    udp_smallest_nonpriv_port 2050
ndd -set /dev/tcp \
    tcp_smallest_nonpriv_port 2050
```

- SYN time-out. A good example of Titan's use as a short-term workaround until a security patch has been disseminated by the vendor. This script was produced from a CERT advisory and placed into a Titan module to run on all local systems to reset system parameters to a safer level until the vendor was able to produce a permanent fix (still useful on older systems!):

```
ndd -set \
    /dev/tcp tcp_ip_abort_cinterval \
    10000
echo "tcp_param_arr+14/W 0t10240" | \
    adb -kw /dev/ksyms
/dev/mem
ndd -set /dev/tcp tcp_conn_req_max 8192
```

- Ping echo. Titan can set up your system so that it does not respond to broadcast ping requests. Why is this important? Attackers often use a ping flood as a DOS attack. In addition, by turning off response to broadcast echo it makes it more difficult for potential attackers to probe our system by sending a ping -s to the broadcast network address:

```
ndd -set /dev/ip \
    ip_respond_to_echo_broadcast 0
```

Startup files

- /etc/rc.* , /etc/rc?.d/*, etc. The rc shell scripts are full of services which startup at boot time that you may not be aware of. Titan will disable services that can potentially be used to gather system information remotely or aid a potential intruder in an attack – this includes the auto-mounter, the dmi, lpsched, snmpdx, and other daemons. Titan disables these services by either commenting out the services or by simply moving the files from the rc* directories.

Configuration files

- sendmail.cf. Titan enables the privacy flags that were introduced in sendmail version 8 with the “goaway” option (among other things this disables VRFY and EXPN), as well as setting the sendmail logging to a reasonable level:

```
Opgoaway
O LogLevel=5
```

- inetd.conf. Titan tears out many of the default services in the Internet services daemon's configuration file. Most of the daemons installed by default are too chatty, historical sources of system vulnerabilities, and operationally unnecessary. Any inetd service that isn't protected by using tcp_wrappers or otherwise restricted and logged (and/or encrypted) is inherently insecure. You should view any program that talks to the network with grave suspicion.
- ftpusers. If the file /etc/ftpusers exists and has users names listed in it, then those users are not allowed to use ftp. Titan adds in system users such as “bin”, “lp”, “root”, and others.
- nsswitch.conf. The contents of this file can be as dangerous as having a “+” in your /etc/hosts.equiv. Having an “nis” or “nis+” entry in this file gives control of crucial files that your system trusts to a remote system. Titan takes the approach that if the local host doesn't know about a remote system, then that remote system can be a threat. Titan errs on the side of safety and simply builds a minimal /etc/nsswitch.conf file using the /etc/nsswitch.files sample file as a starting point for you to build upon, if necessary.
- syslog.conf. Titan modifies /etc/syslog.conf so

that console auth notice messages also get logged to a file.

File and Directory Permissions

- System umask. Titan forces the system (root) to use a default file creation mask (022) that is more secure than the default. This forces all the system daemons to create files with saner file permissions.
- System files and directories. One of the oldest (and still one of the easiest) ways of bypassing system security is to find a directory or a binary file that a privileged user (root most often) is going to access or execute. If that file or the directory that the file/binary lives in is modifiable by another user, then that user can gain additional privileges. Because of the great number of potential problems and different security models for different types of systems (firewalls, servers, desktops, etc.), Titan has three modules that each repair or change one or more aspects of this. All of them can be run for maximum security.

General System Configuration

- eeprom. One of the few things that Titan simply checks and doesn't fix is if you have “security-mode” set in your EEPROM (we don't fix this because you have to choose a password yourself). Don't let us say “we told you so”! If you don't set your EEPROM password, someone else may set it for you – and halt your system. Then you'll need a new EEPROM (or many of them, if they break into multiple systems!), which can take a significant amount of time, especially if you're running an older or discontinued architecture.
- vold. Vold(1M) may seem innocuous, but letting users mount file systems without being root doesn't sound like a good thing to us, even if the Sun vold doesn't allow SUID root shells on the file system being mounted. You might consider allowing this on desktops, but certainly not on servers or firewalls.

Passwords and Authentication

- /etc/passwd. Titan deletes or disables system accounts that are never (or should never be) logged into. Any user or system accounts left on the system have passwords or are disabled, and system accounts other than root and sys (which starts accounting) also have a special non-interactive shell put in place.
- Network Information Services. Titan disables NIS, NIS+, and DNS for name resolution from /etc/nsswitch.conf. While all of these network naming services are insecure, you might have to enable DNS, although we suggest keeping it off whenever possible. Never run NIS or NIS+

on a secure system – or it won't be.

- loginlog. Titan creates /var/adm/loginlog so that the system will log more than five failed login attempts. (This doesn't work with all services (telnet for example).

Titan and Your Security Policy

Titan creates an /etc/issue file with a dire warning to stay away from your system. The contents of this file appear in front of the login prompt. By default it contains:

```
#####
# This system is for the use of
# authorized users only. Individuals
# using this computer system without
# authority, or in excess of their
# authority, are subject to having
# all of their activities on this
# system monitored and recorded by
# system personnel.
#
# In the course of monitoring individuals
# improperly using this system, or in
# the course of system maintenance,
# the activities of authorized users
# may also be monitored.
#
# Anyone using this system expressly
# consents to such monitoring and is
# advised that if such monitoring
# reveals possible evidence of criminal
# activity, system personnel may
# provide the evidence of such monitoring
# to law enforcement officials.
#####
```

Although we strongly suggest running all (or nearly all) the modules in Titan, we realize that not everyone can afford such strident security measures. Titan thus allows you to run different sets of modules as desired by using a simple configuration file. This configuration, or policy file, is a standard UNIX-style configuration file that uses pound signs (“#”) for comments and contains one Titan module (with any arguments desired) per line. We include two sample files, for potential use on desktop and server systems.

Note that the desktop policy disables *sendmail*(8). Since firewalls often deliver or forward mail, this is an optional Titan module. Desktops and most servers have no business running sendmail, however.

Implementation

As previously mentioned, Titan is a master script that runs a collection of Bourne shell scripts. However, before getting any results or fixes from Titan, you must first run the Configure script, which figures out which OS type and version you're running and creates links to the proper Titan modules. Unless something goes awry, this takes no input from the user.

Once configured, Titan has three primary modes of operation to choose from: Fix, Verify, and Inform.

Fix, the most commonly used, simply tells Titan to run out and fold, spindle, and mutilate your system in all the ways it knows about in order to create a more secure system, while informing you of what it is doing. The Verify mode uses the same set of tests but instead of changing the system it simply informs you that various changes would be made if run in the fix mode. The Inform mode takes no investigative or corrective action, but simply echos the function of each module.

Unless you're using a predefined policy, the main Titan script will run all the programs in the module directory with the same mode. You can either create a policy or simply move any modules that you don't want run out of the module directory and Titan will not run them.

Each Titan module accepts one of three arguments – Fix (-f or -F), Verify (-v or -V), or Inform (-i or -I). The Inform argument merely prints out what the script will do. Unless run under the policy mode, Titan runs the modules in alphabetical order (sorted by the shell using the “*” wild card).

The Anatomy of a Titan Module

Although Titan can run any executable program, it is currently written almost entirely in Bourne shell. The Titan scripts are heavily commented, and were intended to be easy to understand. Following the same general form, they start by setting a safe umask and with the copyright notice:

```
:
#
umask 022
# This tool suite was written by
# and is copyrighted by Brad Powell,
# Matt Archibald, and Dan Farmer
# 1992, 1993, 1994, 1995, 1996,
# 1997, 1998 with input from
# Casper Dik, and Alec Muffett.
#
# The copyright holder disclaims
# all responsibility or liability
# with respect to its usage or its
# effect upon hardware or computer
# systems, and maintains copyright
# as set out in the "LICENSE
# document which accompanies
# distribution.
```

The scripts then set the path and do a sanity check to verify root is running the program (since Titan almost exclusively modifies root or system owned files, it makes little sense to run it as anything else):

```
# who am I?
MYNAME='basename $0'

# UCB rules!
PATH=/usr/ucb:/bin:/usr/bin:/sbin
```



```
# did things work out or not?
action='./sanity_check $MYNAME $1'
if test $? -ne 0 ; then
    exit 1
fi
```

Titan scripts then have three functions – Intro(), Check(), and Fix() – to do all the serious work.

The Check() function used when a Titan script is run in the “-v” (“V” for Verify) mode. You might look through a few of the Titan scripts to see some of the ways the Check() function examines the system. The only action that all Titan scripts do in the Check() function is to output either “PASSES CHECK” or “FAILS CHECK”, so users can figure out if this Titan fix is needed or already applied to the system. It can be as simple as (taken from dmi-2.6.sh); see Figure 1. The fix code is similar, but instead of simply stating that there is a problem, it actually takes action (this code snippet is from disable-L1-A.sh, which disables the L1-A or stop-A keyboard sequence by modifying /etc/default/kbd); see Figure 2.

Finally, a Titan module processes the user arguments to see what action to take, and returns a 0 if the module is successful, and non-zero if something goes wrong.

Building a Titan Module

To build your own Titan module, you might want to start out with the \${TITAN-HOME}/arch/sol2sun4/src/stubs/skeleton script – unless, of course, you want to write something other than in Bourne shell. The key points are that the module accepts the basic three

arguments (-i, -v, and -f) as well as outputting an appropriate message based on the success, failure, or the detection of a problem.

For example, simply create a Perl program called “runme.pl” which accepts the standard Titan arguments (-i, -v, or -f) and put it into the Titan module directory. Running “Titan -f” would cause all the scripts that are in that directory – including your new one – to be executed with the “-f” flag.

It is imperative to keep in mind that if you write a Titan module it will be run as root and probably mangle the system in some fashion. Be careful with the code – it’s easy to disable or otherwise make a system useless with a single errant character or a subtle logical error.

Porting Titan to Other OS’s

Despite the fact that at present Titan only operates fully on Sun Microsystem’s operating systems we feel that Titan could be useful with fairly minimal additional steps on other complex systems. To begin with, the basic framework of Titan runs on both main flavors of UNIX – the UCB (Solaris 1.1) and System V (Solaris 2.x) universes. Perhaps a third of all the Titan scripts would work or will work with minor tweaking on most out-of-the-box UNIX’s. Armed with some basic shell scripting and a bit of security knowledge it would not be difficult to port a significant amount of the original Titan code to most systems. Of course, we would be happy to try to place OS specific code on our WWW site.

```
Check() {
    if [ -f /etc/init.d/init.dmi ]; then
        echo " dmi daemon is enabled: FAILS CHECK"
        exit 1
    else
        echo " dmi doesn't start at boot time: PASSES CHECK"
    fi
}
```

Figure 1: Simple fix application test.

```
if [ -f /etc/default/kbd ] ; then
    echo "      Disabling the abort sequence "
    ed - /etc/default/kbd <<- !
    a
    KEYBOARD_ABORT=disabled
    .
    w
    q
    !
    echo " Modifications to /etc/default/kbd complete"
fi
```

Figure 2: Simple disable script.

One of the most important parts of Titan, however, is its collection of little tricks and techniques that are unique to Solaris. The best place to begin amassing a collection of security tweaks for a different system is with the documentation and WWW site of the vendor of the system involved. Nearly all UNIX's have documentation with fairly good sections on security, and many put out security advisories when new problems crop up on the Internet. The BugTraq mailing list and <http://www.rootshell.com> are also excellent references, and it's usually possible to get the older advisories on-line too. Anything that can be typed in at the command line could be placed in a Titan module, including complete compiled or interpreted programs from any languages (C, perl, python, etc.).

Even Microsoft's Windows NT has the potential to be "Titanized." NT version 5 is supposed to come out with a scripting language based on ksh, and such rudimentary things as deleting all default share values, blocking the default guest accounts, and setting up a meaningful set of password and account management policies could be easily written. We are currently investigating HP-UX and Linux as new platforms.

Conclusions

Host-based security is NOT dead, even in the largest installations. Indeed, as organizations grow larger and their resources drop correspondingly they will be required to pick their security fights wisely – and we feel that Titan can be useful in protecting or evaluating the security of key systems, such as firewalls, production servers and other critical hosts.

Titan has been invaluable to us in our work as security professionals – running Titan improves the security of a system in almost all cases. And while it is not impossible to create tight, well-maintained, secure systems, it is, time consuming and very difficult! And it's easy to miss one or more of the many (sometimes crucial) details. Most professionals end up cobbling together various tools using ad hoc checklists when installing or auditing a system. Titan is well-suited for auditing systems as well as creating automated, formal checklists; if a technical security policy does not exist it can suggest the beginnings of one and either determine or force the adherence of a system to it.

It should be said, as disappointing as this may be, that the creation of a secure system is (and will probably always be) far more involved than simply running a computer program. Without a good security policy that is adhered to and a diligent and conscientious system administrative staff that keeps abreast of the latest security news and issues, Titan is relatively useless.

Titan's development future seems bright. Brad, both the originator and the main force behind the effort, has worked on and used Titan for many years, and has no plans to abandon it now. The other

coauthors will be contributing as well, and while we all hope that the Internet community will give ideas or Titan modules, Titan's future is not dependent on that. Methods for backing out of the changes Titan makes, a simple GUI interface for policy management, and ports to other systems are all currently being investigated.

Security tools, from COPS, the TCP wrappers, Crack, Tiger, SATAN, to the many commercial security tools currently available, are invaluable to keeping systems monitored and secure. We respectfully put forth Titan as another freely available tool in the public security defense arsenal, and hope that it proves as valuable to you as it has been for us.

Availability

The latest version of Titan (currently 3.0) is available at: <http://www.fish.com/security/titan.html>. The authors can be contacted by email at: <titan@fish.com>.

Author Information

Dan Farmer performs security research at Earth-Link Networks, Inc. In past lives he has authored or coauthored various security programs and papers, most notably the COPS and SATAN packages. He also worked for several years at Sun Microsystems with Brad and Matt, and can be reached via email at zen@fish.com.

Brad Powell has been in the Computer and Network Security field for over 10 years. As Senior Security Architect for Sun Professional Services, he designs security solutions including Firewalls, Security Architectures, and specialized security products for banks, industry, and government agencies.

Formerly Brad held the position of Network Security Engineer designing Sun's Firewall, security architecture, and network security policies. Duties also included electronic intrusion detection and prevention, and implementing security solutions on thousands of internal Sun networks, computing platforms, and applications, as well as assisting law enforcement agencies worldwide in investigating computer crime. Brad can be reached via email at Brad.Powell@sun.com.

Matthew Archibald left Sun Microsystems Inc. in 1992, after five years as a systems admin/security engineer. After some extended work outside of Sun Microsystems dealing in building and managing mixed-platform environments he joined Securix/Dynasoft in 1994 as a Security Consultant. Matthew subsequently returned to Sun Microsystems Professional Services for a short stay, providing similar services to international customers. Today he works as the Information & Networks Security Officer for KLA-Tencor corporation in Santa Clara CA. and can be reached via email at ir003355@mindspring.com or Matthew.Archibald@KLA-Tencor.COM.

Bibliography

The BugTraq mailing list. Currently bugtraq@netspace.org.

<http://www.rootshell.com>.

Smashing The Stack For Fun And Profit by Aleph One, Phrack 49, Volume Seven, Issue Forty-Nine, File 14, November 08, 1996.

Sun Microsystems, Inc. System Manuals, Sun Microsystems, Inc., 1988-1998.

Sun Microsystems, Inc. Security Bulletin, Sun Microsystems, Inc., 1991-1998.

Infrastructure: A Prerequisite for Effective Security

Bill Fithen, Steve Kalinowski, Jeff Carpenter, and Jed Pickel – CERT Coordination Center

ABSTRACT

The CERT Coordination Center is building an experimental information infrastructure management system, SAFARI, capable of supporting a variety of operating systems and applications. The motivation behind this prototype is to demonstrate the security benefits of a systematically managed infrastructure. SAFARI is an attempt to improve the scalability of managing an infrastructure composed of many hosts, where there are many more hosts than hosts types. SAFARI is designed with one overarching principle: it should impact user, developer, and administrator activities as little as possible. The CERT Coordination Center is actively seeking partners to further this or alternative approaches to improving the infrastructural fabric on which Internet sites operate. SAFARI is currently being used by the CERT/CC to manage over 900 collections of software on three different versions of UNIX on three hardware platforms in a repository ([/afs.cert.org/software](http://afs.cert.org/software)) that is over 20 GB in size.

Background

Since the formation of the Computer Emergency Response Team, ten years ago, it and nearly a hundred other subsequently formed incident response teams have been providing assistance to those involved in computer and network security incidents. The CERT® Coordination Center has participated in the response to well over 10,000 incidents over that period. Throughout that interval, incident response teams have attempted to convince their constituencies to apply security patches and countermeasures to hosts on a routine basis. Over the course of its life, the CERT/CC alone has issued over 160 advisories, 70 vendor-initiated bulletins, and 20 summaries. Most of these documents urge system administrators to apply a variety of countermeasures. Yet, on a daily basis the CERT/CC receives incident reports involving hosts that have been compromised by intruders exploiting vulnerabilities with publicly available patches or countermeasures. The overwhelming majority of compromises continue to be a result of sites running hosts without applying available countermeasures.

In a recent survey we conducted, the most frequent reasons for continuing to operate hosts without available countermeasures were:

- Insufficient resources
- Difficulty in finding and understanding countermeasure information
- The inability to administer configuration management across a large number of hosts.

At many of these sites, the basic computing infrastructure is an impediment to timely distribution of configuration changes. These sites operate at a substantially higher level of risk than those with solid infrastructures. Such sites are not only more likely to be compromised, but they are also less likely to be able to adequately detect and recover from

compromises. They are often forced to undertake large efforts to secure their hosts and networks. Often the required resources are unavailable, preventing administrators from completely recovering from compromises, usually leading to subsequent compromises.

In this paper, when we refer to an organization's information infrastructure, we mean:

- The organization's installed base of computing and networking hardware,
- The system and application software operating on that hardware,
- The policies and procedures governing the design, implementation, operation, and maintenance of the software and hardware, and
- The people who perform those procedures.

One assumes that, in general, such an infrastructure exists only to serve some business purpose of the organization. In a very real sense, an organization's information infrastructure is business overhead-part of the cost of doing business.

An information infrastructure management system (IIMS) is a system whose purpose is to automate the maintenance of an information infrastructure.

Motivation

A site with an adequate information infrastructure finds itself able to detect and recover from compromises in short order. It can repair or recreate hosts using automated mechanisms designed to deal with such demands. It can also protect itself from many types of attacks through proactive deployment of countermeasures.

A site without such an information infrastructure must spend much greater effort to detect and recover from compromises or to proactively deploy countermeasures. This effort is proportional to the number of hosts being managed. At a small site, this may be

acceptable or even ideal, but tends not to be scalable as the site grows due to resource constraints.

At a site with an effective information infrastructure management system, the effort to detect and recover from compromises or to proactively deploy countermeasures is substantially less. In general, the effort to deploy a change is proportional to the number of different types of hosts, rather than the number of hosts. However, the effort to prepare for such a deployment is higher due to packaging requirements of the IIMS.

Chart 1 shows some simpleminded math to illustrate these ideas.

Let:

N_H be the number of hosts

N_P be the number of host platforms

E_H be the effort to make a manual change on one host

E_P be the effort to prepare a change for automatic deployment on any host of one platform

E_D be the effort to deploy a change to one host automatically

Then:

E_M is the total effort to make a manual change on N_H hosts:

$$E_M = E_H \times N_H$$

E_A is the total effort to make a change automatically on N_H hosts

$$E_A = E_P \times N_P + E_D \times N_H$$

Therefore:

The scalability breakeven point is when:

$$E_M = E_A$$

If one assumes that in a good IIMS, the effort to deploy a change on one host automatically (E_D) is negligible, then the breakeven point is when:

$$N_H \times E_H = N_P \times E_P$$

Chart 1: Mathematical derivation of fundamentals.

As you can see from this simplified model, at the breakeven point, the effort to deploy change manually is proportional to the number of hosts, while the effort to deploy the same change is proportional to the number of host platforms (types of hosts).

Requirements

The CERT Coordination Center is currently developing an experimental IIMS called the *Securely Accessible Federated Active Repository for Infrastructures* (SAFARI). Via SAFARI, the CERT/CC is pursuing two primary goals:

- Enable sites to effectively, securely, and economically distribute host-resident software from one or more managed repositories.
- Enable sites to manage all software throughout its entire operational lifecycle focusing on quality, reliability, and integrity.

SAFARI is being designed to meet four primary objectives:

- Construct each host in its configured state automatically from an empty disk with minimal manual intervention.
- Reconstruct each host to its configured state after a security compromise or catastrophic failure automatically with minimal manual intervention.
- Upgrade each host with new applications, operating systems, patches, and fixes automatically on a regular basis with no manual intervention.
- Maintain each host in its configured state automatically with no manual intervention.

In order to achieve the desired influence in the system administration community, three additional secondary objectives must be met:

- It should be engineered for an extremely high host-to-administrator ratio.
- It should facilitate the sharing of software among multiple, not necessarily mutually trusting, administrative domains.
- It should follow a policy-based model that works with a wide range of organization sizes.

Whatever compromises are necessary throughout the evolution of SAFARI, the following constraints must not be relaxed:

- Guarantee that software is distributed onto authorized hosts only.
- Guarantee that software is delivered with integrity to each host and that it is installed correctly.

The design should be guided by the following overarching philosophy:

- It should support user, developer, and administrator activities in a manner that most closely parallels those same activities before SAFARI.

Design Assumptions

This work differs from related projects in some ways. While there have been many projects relating to large scale host administration or software distribution, this project is motivated primarily by scalable security. As a result, we have made certain design assumptions that run counter to prior published work.

We don't consider disk space conservation to be a significant design motivator. Our most recent acquisitions of reasonably inexpensive 18 GB disks supports our assumption. Therefore, a number of space conservation techniques can be immediately discarded. For example, the process of preparing software for distribution via SAFARI is complex enough without an extensive set of space saving rules to follow. Therefore, we decided not to have such rules. Unlike systems that divide a software package between platform-independent and platform-specific [20], we decided that minimal impacts on the build and installation process would easily pay for the

additional disk space. Another impact of this assumption is that everything that goes into building a unit of deployable software is important and should be preserved. This includes files produced during intermediate steps of a build and install procedure. Such intermediate files can be useful during testing and debugging a deployed unit.

We consider individual host performance to be more important than local disk space. We decided to take the default position, that unless otherwise directed, it is the host administrator's intention to install all distributed software locally. We realize though that it might be necessary to locate certain parts of a local host filesystem on a fileserver.

We believe that software maintenance and distribution should be as automated as possible. As a result, we have decided that every available function within SAFARI must be available via a shell command; no GUI only capabilities are allowed. At the appropriate time, we will introduce GUI capabilities to reduce the complexity of the system, but not before a reasonably complete set of functions are already available.

We believe that end-to-end repeatability is of great importance. Repeatability and security are very closely related; security cannot be maintained without repeatability. Therefore, we decided to define the SAFARI model to include package construction activities in SAFARI; this increases the probability of being able to either repeat a build process or to audit it--tracing binaries back to source.

SAFARI, like virtually all similar systems, is focused on the management of small, cohesive, largely independent **software units** (SU). Unlike other systems, SAFARI manages four types of SU's, each useful for a different phase of the software operational lifecycle. All of these SU's are stored in a relocatable structured filesystem tree, called the **SAFARI Repository**. The matrix in Table 1 shows how the various types of SU's relate to one another.

SAFARI is designed to manage software targeted for multiple platforms in a single repository. Both packages and images are labeled with a platform label. Neither collections nor clusters are platform specific. We will examine each of these in more detail.

As an aside, the current implementation of SAFARI depends heavily on Transarc AFS [8] for a variety of administrative functions. This has two

significant consequences (in the genre of good news/bad news):

- The fundamental architecture of SAFARI is greatly enhanced by basic AFS capabilities (e.g., network authentication, integrity, and security). Consequentially, a wide variety of SAFARI administrative tasks (e.g., space allocation, quotas, access control) are greatly facilitated by the use of AFS administrative capabilities. The capabilities provided by AFS were considered a required architectural foundation for a system on which every host at a site may depend. Any other environment that provides similar functions would be acceptable (e.g., DFS).
- The current implementation of SAFARI is tightly integrated with and dependent on AFS and is therefore useless to sites unwilling or unable to run AFS. Note, however, that SAFARI does not require that a site use AFS for anything other than housing the SAFARI repository.

Platforms

Being a multi-platform information infrastructure management system, SAFARI must define and manage the concept of a **platform**. Attempting to be as flexible as possible, SAFARI makes few assumptions regarding characteristics of platforms or the relationships among platforms. In SAFARI, platforms are assigned arbitrary labels by SAFARI administrators as they are needed. SAFARI platform names are composed of any combination of mixed case alphanumeric characters plus dash and underscore, starting with an alphabetic character; they match the regular expression:

```
/^[a-zA-Z] [-a-zA-Z0-9_]*$/
```

SAFARI assigns no meaning to the contents of the name; SAFARI administrators are free to choose any platform naming convention they like.

Each SAFARI host must be assigned a platform name. Typically, a host platform name refers to a specific combination of hardware and system software. For example, Sun Solaris 2.5.1 running on a Sun Ultra 2 model 2170 is a platform that one SAFARI site may assign the label *sparc-4u-sunos-5.5.1*; another may assign *solaris-2.5.1_sparc4u*, and a third may choose *sun4u_251*.

	Source Software Units	Deployable Software Units
Development of Independent Software Units	Collection	Package
Configuration or Integration of Software Units	Cluster	Image

Table 1: Relationships between software units.

Each SAFARI deployable software unit (package or image) is also assigned a platform name. Deployable software unit (DSU) platform names are typically more abstract than host platform names. For example, a PERL script might easily be written so as to run on Solaris 2.5.1 on SPARC and Red Hat Linux 5.1 on Intel. In which case, one might choose to assign to the package containing the PERL script a platform name such as *unix*. Choosing an appropriate platform name for a DSU can sometimes be challenging. Continuing this example, suppose that the PERL script did not work on AIX 4.3 on PowerPC. What platform name would be appropriate then?

All platform names are recorded in the SAFARI repository database. In addition, the repository contains a mapping between each host platform name and the platform names of DSU's that are allowed to run on hosts of that platform. That is, it documents which DSU's can be run on which hosts by platform name. For example, the repository might declare that a host platform named *sparc-4m-sunos-5.5.1* can run packages or images tagged with platform names *sparc-4m-sunos-5.5.1*, *sparc-sunos-5.5.1*, *sparc-sunos-5*, *sunos-5*, *sunos*, *unix*.

All SAFARI tools that deal with platform names require that platforms be recorded in the SAFARI repository database.

Packages

The SAFARI **package** is the easiest class of SAFARI software unit to understand. SAFARI packages are logically equivalent to the installable units in many other software management systems: **packages** in Sun Solaris [1] and UNIX System V [2], **packages** in Red Hat Linux [3], **packages** in IBM AIX [4], and many others. In fact, the design for SAFARI packages was taken almost directly from CMU Depot [5] (an ancestor of SAFARI, *Alex*, developed at the University of Pittsburgh [6] actually uses *depot* internally).

A SAFARI package is a sparse relocatable filesystem tree. It has exactly the same internal directory structure as a host at which it is targeted for installation. That is, if a given package is intended to install the files */etc/hosts* and */usr/bin/cp*, then one would find within the package filesystem tree the files *etc/hosts* and *usr/bin/cp*. There is currently no canonical transportable format for a SAFARI package (such as *tar* or *RPM*). Within each package is a special directory which contains meta-information about the package and is not intended to be installed on the target system. This directory is currently named *depot* (more ancestral evidence). It contains a manifest of the package's contents and a certificate of authenticity, digitally signed by the person who produced the package. SAFARI currently uses PGP 2.6 [10] for digital signatures and MD5 [11] for message digests of files described by the manifest.

It may be obvious from the above description, but SAFARI packages are *platform dependent*. Each package is labeled as being applicable to (installable on) a particular platform.

SAFARI provides tools to aid both the SAFARI administrator and the SAFARI collection manager in the creation, population, release, integration, testing, deployment, deprecation, withdrawal, and destruction of packages in a repository and on client hosts.

In most software management systems, package-like entities come from outside the software management system. In SAFARI, a package is derived from another software unit, a SAFARI **collection**.

Collections

The easiest way to understand SAFARI **collections** is to think of them as package sources. One set of sources targeted at *n* platforms requires one SAFARI collection from which *n* SAFARI packages are produced. SAFARI collections are structured as filesystem trees, similar to packages, but in contrast to packages, have minimal mandatory internal structure. This flexibility is required to support the wide structural variety of open source systems. There are three generic areas (subdirectories) within a collection:

- **build**. This area is used to house collection source code, makefiles, etc. There is no preferred structure within this area. Collection managers are free to establish their own conventions for structure in this area. For example, in our repository, it is typical that a *tar.gz* file for sources resides in the *build* directory with the expanded *tar.gz* file in lower directories. The extensive use of RCS [12] or SCCS [13] is strongly encouraged, but remains a collection manager choice.
- **OBJ**. This area is used to house compiled pre-installation binaries (e.g., *.o, *.a) for each platform. The OBJ tree is expected to be managed by a SAFARI tool named *pf* (described below), but may be used as the collection manager sees fit. The *pf* command establishes a parallel tree of symbolic links for each platform rooted in the second level of the OBJ tree that mirrors the *build* tree. Those familiar with the use of the *Indir* command in the X11 distribution [7] to accomplish multi-platform builds can imagine how *pf* works.
- **META**. This area is used by SAFARI to hold meta-information about the collection. For example, the mapping between build platforms (the platform on which the build is performed) and installation platforms (the platform on which the package is expected to run) is contained in *META/platforms.map*.

In addition, for the sake of convenience to the collection manager, any existing unreleased packages

are available under the following area within a collection.

- **DEST.** This area contains one directory for each existing unreleased package for the collection. Technically, these directories are not part of the collection. They are AFS volumes mounted here to make the build process easier for the collection manager by allowing the installation destination to be specified using short relative (and therefore relocatable) paths in makefiles.

Of course, many packages are only available in binary, platform-specific form. For such packages, SAFARI provides tools that can create a package from a delta of a manual installation host. This idea came from the WinINSTALL [9] tool which does the same thing for Microsoft Windows operating systems. It takes a snapshot of an existing host before some software is installed. After the installation, the tool compares the before and after states of the host to produce a SAFARI package that when deployed via SAFARI results in the same host changes as the software installation steps. This approach can be used for any arbitrary modifications, but is better left for situations where software must be installed via an installation procedure that cannot easily be reverse engineered. For some platforms (i.e., Sun SunOS 4), this may be the easiest way to capture patches for distribution to other hosts via SAFARI. Repeatability using such a tool capturing manual steps is poor. Fortunately, this tool is rarely needed since most software installation procedures allow for one to install wherever one wants.

Clusters and Images

In cases where a collection and its packages do not depend on other packages or local configuration conventions to operate properly when installed, SAFARI host managers can select which packages they wish to use with little difficulty. An example of such a collection might be GNU CC. GNU CC really only depends on having the proper operating system in place to function properly. Any host manager who was interested in GNU CC could safely just select a GNU CC package appropriate for the host's platform and be highly certain that it will deploy and operate properly.

However, relatively few collections depend so little on other collections and have no dependence on local configuration conventions. For example, the collection that contains the SAFARI software itself, being written in C++ and PERL, depends on a variety of other collections being available to be able to operate properly. The process of selecting, ordering, integrating, configuring, and testing a set of interdependent packages can be complex. In addition, it occasionally happens that combining particular packages requires significant configuration to get them to work correctly.

Infrastructure: A Prerequisite for Effective Security

Facilitating this process is the purpose of SAFARI **clusters** and **images**. A SAFARI cluster is structured and managed exactly like a collection; images are produced from clusters exactly like packages are produced from collections. Like collections, clusters are not tagged with a platform name; like packages, images are tagged with a platform name.

The principle distinction between collections (and their packages) and clusters (and their images) is that clusters produce images that reflect local configuration conventions, while collections produce packages that are intended to be deployable on any host in the world. The normal progression is to build one or more packages from collections with no local assumptions about deployment hosts and then build a cluster which integrates and configures the packages according to local conventions to be deployable on local hosts. Others not following those conventions can build clusters in the same or a different repository according to their conventions using the same packages, thereby, reusing the existing packages.

A cluster can be thought of as a set of collections, integrated together, and its images can be thought of as the set of packages from those collections integrated together. The primary purposes of clusters and images is to reduce the effort expended by host managers trying to integrate packages together. An image is a pre-configured set (including a set of one) of packages that can be selected as a single unit for deployment.

Two reasons for this approach of separation based on configuration information are:

- **Technical:** This separation facilitates the construction of collections/packages that can be readily used at multiple sites. That is, the collections and packages thus created can be easily shared between SAFARI repositories by concentrating configuration information, which is typically very local, in clusters and images. Thus collaborating sites can easily share packages without extensive human interaction. The clusters, presumably, reflect local configuration conventions and must be reengineered from one repository (domain of local configuration conventions) to another.
- **Organizational:** Some persons are better at building packages one at a time and others are better at integrating a group of packages together. This approach allows staff to be applied more effectively to the tasks to which they are better suited. Explicitly separating the integration and configuration step from the development step allows the developer to concentrate on producing a higher quality package faster. Developers focus on the internals of a package; integrators focus on the world around a package.

The Repository

The SAFARI repository houses:

- Collections and their packages
- Clusters and their images
- Database
 - Meta-information about collections, packages, clusters, and images
 - Policy settings
 - Database meta-information

The SAFARI repository is a relocatable filesystem tree. For current SAFARI commands, its location is specified by the `REPO_ROOT` environment variable. Future versions of SAFARI will incorporate inter-repository capabilities.

Collections, Packages, Clusters, and Images

Each different class of SAFARI software unit has its own area in a repository.

- **collection.** The collection directory is the parent directory of all collections. Each collection is housed in a separate subdirectory. As described above, each collection holds four internal areas (META, build, OBJ, and DEST). As an example, the source to the SAFARI version 1.0 collection might, depending on collection names chosen, be found somewhere in `collection/cert.org/safari/1.0/build/`. The SAFARI administrator *safari collection create* subcommand creates the appropriate AFS volumes (for each area), creates the appropriate AFS protection groups, mounts the volumes in the collection directory tree, sets access controls and quotas, and transfers subsequent control of the collection to the collection's managers to manage the build, release, and test steps.
- **package.** The package directory is the parent directory for all released packages. Unreleased packages are AFS volumes mounted under the collection's DEST area only. After completing the build and DEST area installation steps, the collection manager uses the *safari package prepare* and *safari package seal* subcommands to collect meta-information about the package and certify the contents of the package as suitable for release (at least in the mind of the collection manager). The SAFARI administrator then uses the *safari package release* subcommand to transform the package into the correct form for deployment (e.g., set owners and groups according to policy) and mount it in the appropriate place below the package directory. Released packages are housed in directories below package with naming conventions that parallel the collection directory. For example, revision 10 of the SAFARI version 1.0 package for platform *sparc-sunos-5* can be found under `package/cert.org/safari/1.0/10/sparc-sunos-5/`.

- **cluster.** The cluster directory is the parent directory of all clusters. The structure below cluster is exactly the same as that below `collection`. The safari cluster subcommands parallel the safari collection subcommands.
- **image.** The image directory is the parent directory of all images. The structure below image is exactly the same as that below package. The safari image subcommands parallel the safari package subcommands.

Each collection and each cluster has a unique name (i.e., collections and clusters have separate name spaces and can therefore have the same name). Packages and images inherit their name from the collection or cluster from which they were derived. Clusters and collections are named using the same syntactic and semantic conventions. A cluster or collection name is composed of three parts separated by forward slashes:

- **Authority.** The authority is the organization or person who is officially responsible for the files found in the cluster's or collection's build area. For example, the officially responsible organization for GNU Emacs is the Free Software Foundation. For the Red Hat Package Manager, it is Red Hat, Inc. The authority is encoded as the closest possible representation of the organization or person as an appropriate Domain Name Service Start of Authority record name. This means that the syntactical requirements for the authority part are exactly the same as those for a domain name [17]. In the preceding two examples, Free Software Foundation would be encoded as `fsf.org` and Red Hat, Inc. would be encoded as `rpm.org`, since RPM developers have registered an organizational name just for that software. The rationale for including an authority in a collection or cluster name rather than making it an internal attribute of the cluster or collection was to avoid naming wars. Sometimes there are multiple possible authoritative sources for Internet free software; including the authority in the name avoids the problem of requiring a SAFARI administrator to arbitrate which source is more authoritative.
- **Common name.** The common name is the name by which the collection or cluster is commonly known. It is typically exactly the same name as that assigned by the authority. The only restrictions on this part of the collection or cluster name is that it comply with the POSIX requirements for a file name (a single component of a path) [18]. For example, GNU Emacs might be reasonably assigned the common name "emacs" or "gnu-emacs."
- **Version.** The version is the version string by which the collection or cluster is commonly known. It is typically exactly the same version string as that assigned by the authority. The

only restrictions on this part of the collection or cluster name is that it comply with the POSIX requirements for a file name [18]. For example, GNU Emacs version 19.34 would likely be assigned the version "19.14," while Samba version 1.1.18p12 would likely be assigned the version "1.1.18p12."

Thus the full name of the GNU Emacs 19.34 collection might be `fsf.org/gnu-emacs/19.34`.

Meta-Information about Collections, Packages, Clusters, and Images

The SAFARI repository database is a directory of ordinary UNIX files, formatted so that the SAFARI administrator can make manual changes in emergency situations. As such, the database is neither highly parallel nor impervious to failed transactions. The database is protected against concurrent access via a simple lock file mechanism. All of the meta-information and policy files are under RCS control to provide assistance during a manual recovery as well as extensive historical information. For reference, our `db/packages` database is at RCS revision 1.3729 as of 8/24/1998 having started at revision 1.1 on 1/27/1997 (yes, 3,728 revisions).

The repository database is located in `$REPO_ROOT/db`.

- `db/collections`. The collections database contains the mapping between each collection name and its sequence number. Sequence numbers, which can never be safely reused, are used to form AFS volume names. The presence of a record in this database implies that the collection filesystem tree is present in the repository for that collection (the *safari repository check* subcommand reports differences between the collections database and collection filesystem). The use of sequence numbers is intended to deal with severe name length restrictions in AFS volume names (and potentially other underlying technologies that might be supported by SAFARI in the future). Through this indirection, SAFARI supports long collection names that greatly increase understanding in casual SAFARI users (e.g., single-host host managers). Even SAFARI aficionados can benefit from long clear collection names. Our repository currently has over 900 collections; clear naming really is a requirement. When collections are destroyed, their sequence record is moved to the `collections.destroyed` database for historical reference. The content of this database is managed by the *safari collection* subcommand.
- `db/collections-families`. The database of collection-families is used to determine the precedence among a set of collections implementing different major versions of the same software. Each record defines a

family of collections from the collections database. The purpose of this database is to deal with collections whose name changes in unpredictable ways as they evolve. In particular, it is rarely the case that the next version number of a given piece of software can be predicted from the current one. The Linux kernel [15] version numbering scheme is a good example: how should the following version numbers be ordered: 2.0, 2.0.1, 2.0.10, 2.0.100, 2.0.9? Because we know the convention, we know that the proper order is: 2.0, 2.0.1, 2.0.9, 2.0.10, 2.0.100. But we have all seen numbering schemes more unpredictable than this. Each collection family explicitly defines the ordering among a set of collections. The package deployment selection mechanism (see *parcel* below) will automatically choose the latest collection with an appropriate platform and status for the given host.

- `db/packages`. The packages database contains one record for each existing package. Each package is named for the collection with which it is associated. In addition, each package is assigned a monotonically increasing (for each collection) revision number and a platform name. The first package created from a collection is assigned revision number 1. It is expected that two packages of the same collection with the same revision number (but different platform names) implement the same functionality when deployed. This revision synchronization convention makes it easier for a host manager who is managing multiple hosts of different platforms to configure them to provide the same functions to their users. Lastly, each package is assigned a status tag. Status tags are fully described in `db/lifecycles` under *Policies* below.
- `db/clusters`. The clusters database contains the mapping between cluster names and sequence numbers. The clusters database has the same format as the collection database and serves the same purpose for clusters as the collections database has for collections.
- `db/clusters-families`. The cluster-families database is used to determine the precedence among a set of clusters configuring different major versions of the same software. It has the same format as the collection-families database and serves the same function for clusters that collection-families serves for collections.
- `db/images`. The images database contains one record for each existing image. It has the same format as the packages database and serves the same function for images that packages serves for packages.

Policies

A variety of functional policies have been externalized from code into policy specification files. Future versions of SAFARI will expand the use of external policy specifications. There were two purposes behind this decision. First, we thought it likely that many of these policies might vary between sites. Configuration files are easier to change than code. Second, we were ourselves not sure what our policies should be. Moving these policies from code into configuration files has allowed us to easily experiment with alternative policies and to evolve policy over time.

- **db/allocation.** The allocation configuration file sets the policy for how AFS volumes are named and where they are allocated and replicated. There are currently six types of AFS volumes managed by SAFARI, which are initially mounted at the mount points shown in Table 2.

The allocation policy allows one to select AFS volume naming conventions using printf-like “%” syntax to conform to local AFS volume naming conventions. It also allows one to define AFS filesystems and partitions that are to be considered as candidates for holding each type of volume. One can also specify which volumes, if any, are to have read-only replicants and where they can be located. The safari command uses this policy to guide its creation of

AFS volumes. The server selection portion of this policy is only enforced at creation time; volumes can be moved after allocation however a site wishes.

- **db/lifecycles.** The lifecycle configuration file sets the policy for lifecycle phases through which packages and images can pass. The policy defines each phase by assigning it a label called the **status**. Each status label is an opaque string that SAFARI uses for tracking the lifecycle of packages and images. The special status label **unreleased** is built-in and is automatically assigned to packages and images when they are created. When a package or image is released by the *safari package release* or *safari image release* subcommand respectively, the SAFARI administrator must provide a status label to be assigned to the newly released package or image as the last step of the release process. The status labels are recorded in the *db/packages* and *db/images* databases for each package and image respectively. The lifecycle policy governs which lifecycle phase transitions are allowed by means of a transition matrix. The matrix is composed of cells that contain permit/deny indicators for each possible starting and ending status label.

Using this mechanism, a site can define a complex lifecycle through which packages and

	Creation Mount Point (below \${REPO_ROOT})	Released Mount Point (below \${REPO_ROOT})
collection build volume	collection/collection/build	
collection OBJ volume	collection/collection/OBJ	
package volume	collection/collection/DEST/platform	package/collection/ revision/platform
cluster build volume	cluster/cluster/build	
cluster OBJ volume	cluster/cluster/OBJ	
image volume	cluster/cluster/DEST/platform	package/collection/ revision/platform

Table 2: Mount points.

Status Label	Status of Package or Image
alpha	Is being tested in operation on a real machine by its manager; no other hosts should try to run it.
beta	Is being tested by end-users on a small set of beta test hosts.
gamma	Is production quality and can be run anywhere the function it provides is useful.
deprecated	Is being withdrawn for some reason (typically because it is being superseded).
obsolete	Has been withdrawn and no host is allowed to use it anywhere.
destroyed	Has been destroyed (deleted).

Table 3: Status labels.

images must progress from creation to destruction. For example, in the CERT repository, we have defined a series of status labels as shown in Table 3.

We define our lifecycle transition matrix to force packages and images to evolve from **alpha** toward **destroyed**; we do not allow transitions in the other direction. We do, however, allow phases to be skipped (e.g., to **gamma**, **beta** to **obsolete**, **unreleased** to **beta**).

- **db/platforms.** The platforms configuration file sets the policy for naming platforms. Every platform supported by a repository must be defined in this configuration file. Packages and images are labeled with platform names to indicate on which hosts they were intended to be deployed. For any given collection or cluster, packages or images with the same revision number and differing platform names are expected to implement the same functionality. Each platform defined in the platform policy is assigned a unique (across all platforms) small integer to be used in naming AFS volumes. This number is used in a way similar to the collection and cluster sequence number--to reduce the length of the AFS volume name without loss of information. Once a platform is assigned a number, the number can never be reused again (unless one can somehow guarantee that there are not now and never were any AFS volumes created using it).
- **db/platform-families.** The platform-families configuration file sets the policy for how platforms, defined in **db/platforms**, relate to one another. Each platform family is, in effect, a sequence of synonyms. The first platform in each sequence is used as a key and serves as the name of the family. It is expected that the platform name assigned to every host is also the name of a platform family. The deployment mechanism

(see **parcel** below) uses the host platform name as a key to locate the sequence of synonyms for that platform name from the platform family policy. The deployment mechanism then considers any package (or image) that has been labeled with any of the synonyms as a candidate for deployment to the host. Ambiguities are resolved by selecting the first candidate encountered in the order of the synonyms in the sequence.

Using the platforms policy and the platform families policy, one can create a taxonomy of platform names. Consider Figure 1 as a platform taxonomy.

The leaf nodes would then be defined as platform families and the synonym sequence would be all nodes encountered along a directed path from the leaf to the root. Then the platform family *sparc-4m-sunos-5.5.1* would be defined as the platform sequence: [*sparc-4m-sunos-5.5.1*, *sparc-sunos-5*, *sunos-5*, *unix*, *generic*]. Therefore, when the deployment mechanism was activated on a *sparc-4m-sunos-5.5.1* host, it would search for *sparc-4m-sunos-5.5.1* packages (or images) first, followed by *sparc-sunos-5*, *sunos-5*, *unix*, and *generic*.

- **db/protections.** The protections configuration file sets the access control policy for released packages and images. It controls file and directory ownerships, UNIX modes, and AFS access controls lists. It does not control local file system access control lists. This configuration file sets the global policy that governs all packages and images. In addition, each package and image can supply its own policy, overriding the global policy, in **depot/protections** in the package or image. The protections policy provides a default value for AFS access control lists, file ownership, file group, and positive and

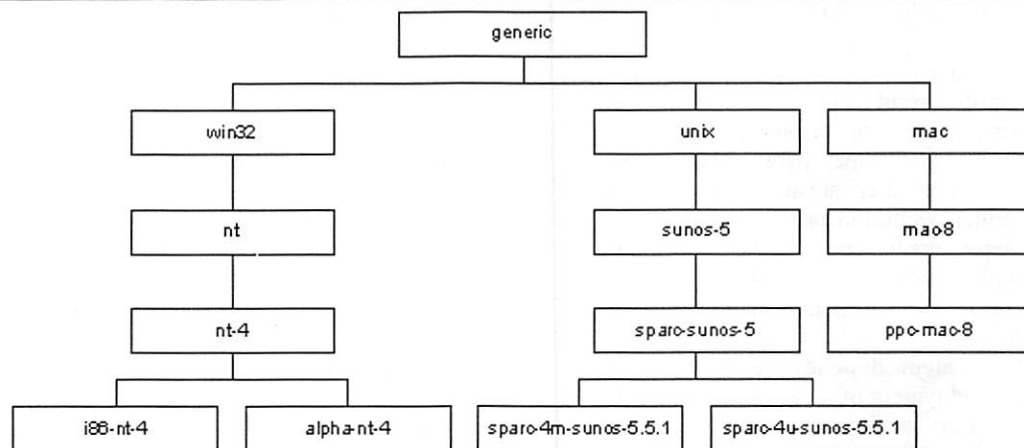


Figure 1: Platform Taxonomy.

negative mode masks. Owners and groups can be specified as names or IDs, but names are preferred (being less platform dependent). The positive mode mask is a mask of mode bits that are unconditionally OR'ed on at release time. The negative mode mask is a mask of mode bits that are unconditionally AND'ed off at release time. Not all mode bits need to be included in either of these masks. If these masks overlap (affect the same bits) the result is undefined.

It is not typically the case that a package or image will override the protection defaults from the global policy, although that is permitted. More often, the package or image protections policy will specify particular paths for which ownerships, modes, or ACL's should differ from the default. This approach is required since collection managers are not typically AFS administrators and as such do not have the privilege necessary to actually set file ownerships or groups. The package or image protection policy is used to convey the collection manager's intent regarding protections to the SAFARI administrator for the *safari package release* or *safari image release* process. Currently, collection managers can request any protection policy they want; it is up to the SAFARI administrator to recognize that a particular release request ought to be rejected due to a suspicious protections policy (e.g., making `/bin/sh set-UID root`).

- `db/pubring.pgp`. The `pubring.pgp` configuration file contains the PGP 2.6 public keys of all persons that can digitally sign something in SAFARI. This is typically collection and cluster managers as well as SAFARI administrators.
- `db/relations`. The `relations` configuration file sets the experimental policy regarding relationships between and among collections, packages, clusters, and images. As of this writing, this policy is still being defined. The intention of this policy is to specify the following relationships:
 - **Conflict resolution** is necessary when two packages or images both wish to deploy the same path. The current deployment mechanism has an implicit conflict resolution policy, but it is considered inadequate for large complex configurations. The depot [16] conflict resolution mechanism is also regarded as too weak.
 - **Deployment dependence** occurs when the deployment of a package depends on the prior deployment of some other package. This is typically only the case when SAFARI related packages are

deployed.

- **Operational dependence** occurs when a package depends on the concurrent deployment of another package in order to be able to function. The deployment mechanism must be able to follow these dependencies and either automatically add missing dependencies or produce diagnostics regarding missing dependencies.
- **Construction dependence** occurs when a collection depends on the prior deployment of another package in order to be built (to produce a package). This is mostly a documentation issue for collection and cluster managers, but plays a role in version and revision tracking. Once in a while it happens that it is necessary to restore a destroyed collection and rebuild it for some reason. To be able to get back to a known state, one must know not only what the state of the collection in question was at that point in history, but one must also know the state of all of the packages that the collection depended on during the build process. We have several times found ourselves in the state where we knew for certain that a given package was built from a particular set of sources, but we could not reconstruct what packages that collection depended on during its build process. This typically happens when the build process of one collection uses a `lib*.a` file provided by another package.
- **Mutual exclusion** occurs when the deployment of one package or image degrades or destroys the ability of another concurrently deployed package or image. The deployment mechanism must be able to detect and avoid this situation.

Experimentation with representation of relationships has been underway without a clear resolution for almost a year. This turns out to be a hard problem.

- `db/roles`. The `roles` configuration file sets the policy regarding who can do what. It defines a set of roles using UNIX or AFS groups. That is, rather than using UNIX or AFS groups throughout SAFARI to authorize operations, SAFARI uses a set of roles for authorization and determines who is in each role once at program initialization time. This allows a site to define the set of UNIX or AFS groups they wish to use to authorize SAFARI operations. There are no predefined roles. A site may define as many or as few roles as their

organizational structure justifies. For each SAFARI operation, the roles policy defines which roles can perform that operation. Each operation also defines who (in the form of an email address) should be notified about each operation.

- **db/structure.** The structure configuration files sets the experimental policy regarding file system structure for release validation. To reduce the possibility of totally invalid file system structures being deployed, even in testing, and destroying a machine, the release process must be expanded to perform a variety of conformance tests on release candidates. For example, if some package accidentally provided a file named `/usr/lib` most hosts would cease to function shortly after that file was deployed. The structure policy is the first of several policies aimed at release conformance tests.

Database Meta-Information

- **db/databases.** The databases configuration file specifies the actual location of all of the above databases. The paths shown above are the ones defined in the CERT SAFARI repository, but any of these databases can be renamed or relocated. In the current implementation, we don't recommend this as it has not been tested. It is possible that some lingering path names may be embedded in code somewhere.
- **db/ hashes.** The hashes configuration file contains the official MD5 hashes of every database defined in `db/database` as of the end of the last database update transaction. These hashes are used as a high performance mechanism for database consistency checking by the deployment mechanism (since it cannot actually lock the database).

Programs

SAFARI includes a number of programs and scripts. Some are provided for repository administrators, some for collection managers, and others for host managers. Even though end-users should be completely unaware of the existence of SAFARI, one tool is provided that may be useful to them.

safari

The *safari* command is the principle tool for repository administrators. Nearly all of its functions are aimed at administration and maintenance of a SAFARI repository. The *safari* command presents a simple object-oriented view of the repository. The repository is divided into several classes of objects and a set of operations that apply to each class. Each class and its applicable operations are presented below.

Infrastructure: A Prerequisite for Effective Security

Creation

```
safari {collection | cluster} \
  create name \
    -manager principal... \
    [-quota blocks]
```

This subcommand creates a new SU, either a collection or a cluster, assigning it a unique new name and one or more principals as managers for the SU. Assignment of multiple managers is not discouraged, but neither is it facilitated in any way by SAFARI. Multiple managers must coordinate their activities regarding the SU. It is common to assign two managers for each SU, one as primary and one as backup. The primary manager has the responsibility to keep the backup manager informed enough so that the backup manager can act in absence of the primary manager (usually this means fixing a bug and releasing a new DSU).

```
safari {package | image} \
  create name \
    -platform name \
    [-revision integer] \
    [-quota blocks]
```

This subcommand creates a new DSU, either a package or an image, which is targeted at a particular platform. When managing multiple platforms at different times, it is sometimes impossible for *safari* to correctly choose a revision number since *safari* cannot know what the manager intends the DSU to contain. Therefore, an explicit revision number can be specified.

Examination

```
safari {collection | cluster} \
  list regexp
```

This subcommand lists SU's by matching a (PERL) regular expression against the names of all known SU's.

```
safari {package | image} \
  list [name] \
    [-platform name] \
    [-status tag] [-managers] \
    [-sizes]
```

This subcommand lists DSU's by matching a regular expression against the names of all known DSU's, by target platform, by status, or some combination of the above.

Preparation for Release

```
safari {package | image} \
  prepare name \
    -platform name
```

This subcommand produces the DSU's depot/MANIFEST file, listing the complete contents of the DSU. Cryptographic hashes (MD5) are used to describe the contents of all files. The manifest does not describe itself. The manifest is constructed in

such a way as to include the intended modes and ownerships of files and directories, rather than their current modes and ownerships. The manifest is used in the release process (below) to actually change the modes and ownerships before releasing the DSU. This allows non-privileged managers to ask for modes and ownerships they would not normally be allowed to set themselves.

```
safari {package | image} \
    seal name \
    -platform name \
    [-user pgpuser]
```

This subcommand produces the DSU's depot/AUTHORITY file, which is a certificate of authenticity for the DSU. The certificate includes a cryptographic hash (MD5) of the manifest, as well as other meta-information, and is digitally signed. In this way, the entire DSU is sealed under the authority of the manager and cannot be altered from the configuration specified in the manifest file without detection. All prospective users of the DSU can see who certified the DSU's contents and decide for themselves what level of trust to accord it.

Validation

```
safari {package | image} \
    check name \
    -platform name \
    [-revision integer]
```

This subcommand validates the content of a DSU. It checks the digital signature on the DSU certificate (depot/AUTHORITY), uses the certificate to check the contents of the manifest (depot/MANIFEST), and uses the manifest to check the contents of the DSU. All checks must match exactly for validation to succeed. Any deviation is reported.

Release

```
safari {package | image} \
    release name \
    -platform name \
    -status tag
```

This subcommand transforms an unreleased DSU into its released form, mounts the AFS volume in its correct location, and updates the repository database regarding the status change for that DSU. The transformation process includes validation of the contents of the DSU (except for modes and ownerships), alteration of modes and ownerships to match the manifest, and alteration of AFS ACL's according to the protection policy in effect for the DSU (global policy + DSU policy).

It is expected that the status assigned to a newly released DSU represents semantics of minimal trust. For example, we use status 'alpha' to indicate that no one except the SU manager should trust the DSU's contents. This prevents accidental deployment of newly released DSU's onto production quality hosts.

Lifecycle Management

```
safari {package | image} \
    setstatus name \
    -platform name \
    -revision integer \
    -status tag
```

This subcommand changes the status label associated with an already released DSU. This is the mechanism that a manager uses to signal his intent that others can begin to trust the contents of the DSU. Multiple levels of trust may be appropriate at a given site. In our repository, we use status beta to mean that a DSU can be deployed onto certain hosts which have been designated for end user functional testing. This deployment automatically occurs, announcements are sent out, and users try out the newly released software on the beta hosts. After the manager receives feedback, he can decide to promote the DSU into full production use (status **gamma** in our repository) or out of service (status **deprecated** in our repository) to be replaced by a new revision.

Repository Management

```
safari repository lock
safari repository unlock
```

These subcommand can be used to manage the reservation of the repository database for extended periods of time. The typical use of this function is to perform several related actions that should be seen by repository users all together, such as releasing multiple interdependent packages at one time. If a host in the process of being constructed were to select some, but not all of the interdependent set of packages for deployment, a variety of inconsistency-related failures may occur (e.g., shared library skew).

```
safari repository check
```

This subcommand is a maintenance function, typically run regularly by a SAFARI administrator to ensure that the repository database is syntactically correct and that the database actually reflects what is installed in the repository, and vice versa. In addition to detecting several different kinds of inconsistencies, this subcommand can propose (for certain kinds of inconsistencies) corrective actions to be taken by the administrator. Inconsistencies between the repository and the repository database almost always occur as a result of AFS administrative failures of some sort, lost AFS volumes being the most common.

```
safari repository addkey
safari repository listkey
```

These subcommands are used to manage PGP public keys stored in the repository database. Every administrator and manager must have a PGP public key stored in the database.

```
safari repository showpolicy
```

This subcommand can be used to print human readable forms of a variety of repository policies (described above).

pf

The *pf* command is designed to help the collection manager build binaries for multiple platforms from one set of sources with minimal impact on uni-platform build procedures. The *pf* command is an abbreviation for platform; it is so short because it is used often enough that brevity is valuable. The first and foremost task of *pf* is to determine the name of the target platform for the build process. If *pf* is invoked with no arguments, it simply prints the target platform name on stdout. This is sometimes useful in scripts and makefiles. Since it requires the *platform.map* file in the collection's or cluster's META area to determine the build target platform from the actual platform of the build host, it does not function outside of a collection or cluster.

Pf provides multi-platform build assistance in two ways: platform-specific parallel trees of symbolic links and shell command wrapping. The parallel trees of symbolic links are managed by two command line options: *--update* and *--clean*, typically used together.

The *pf* command also provides multi-platform installation assistance hiding the installation location from the collection manager. This is accomplished via the *--install* and *--purge* options.

The basic syntax of the *pf* command is:

```
pf [--clean] [--update] \
  [--purge] [--install] \
  [shell-command [shell-args] ...]
```

The *--clean* and *--update* options are concerned with maintaining the platform specific trees of symbolic links in the cluster's or collection's OBJ area.

- *--clean* or *-c*: The *--clean* option specifies that any dangling symbolic links in the symbolic links tree are to be removed. Any empty directories are also removed.
- *--update* or *-u*: The *--update* option specifies that before executing any shell command specified (see below), the platform specific tree of symbolic links located in the collection's or cluster's OBJ tree should be updated to match the current directory and all of its subdirectories. Any missing or incorrect symbolic links in the symbolic link tree are corrected to point to their corresponding file in the build tree. Any missing directories are also created.

When *--clean* and *--update* are both specified, cleaning occurs before updating.

The *--purge* and *--install* options are concerned with constructing the platform specific

packages or images mounted in the cluster's or collection's DEST area.

- *--purge* or *-P*: The *--purge* option simply removes all contents of the platform specific package or image mounted in the DEST area. This option is almost always used in combination with the *--install* option to accomplish a completely clean installation into a package or image.
- *--install* or *-i*: The *--install* option simply defines the DESTDIR environment variable to the root of the platform specific target package or image mounted in the DEST area. Many makefiles already expect a variable defined in this manner, making its use completely transparent. For those that don't use DESTDIR, trivial changes in the makefiles are required to support this convention. For example, in software constructed using a recent version of *autoconf* [19], one can define the prefix to support DESTDIR in the following way:

```
./configure \
  --prefix='${DESTDIR}/usr/whatever'
```

The *pf* command is also used to execute every command in the build procedure. After processing the above options, if there is a shell command and optional arguments remaining on the command line, *pf* will change directory to the parallel directory in the symbolic link tree in the OBJ area and the execute the specified command via *execve(2)* [20]. This is easier to see by example. The following example assumes a normal *autoconf* managed piece of software named *fsf.org/something/1.0*:

```
$ cd $REPO_ROOT/collection/fsf.org/\
something/1.0/build/something-1.0
$ pf -cu ./configure \
  --prefix='${DESTDIR}/usr/local'
$ pf gmake
$ pf -Pi gmake install
```

Using this exact same set of commands, one can build this collection for every platform one wishes (assuming the software was written to support the target platforms). As one can see, this is a minimal alteration from the normal uni-platform build procedure.

ft

The *ft* (short for filetree) command is also aimed at simplifying the life of the collection manager. While building Internet-available software from source is desirable, little commercial software is delivered in source form. The *ft* command is designed to help in dealing with software delivered in binary-only form. In particular, it is helpful when such software comes with an obscure installation procedure that is difficult or impossible to coerce into installing properly into the DEST area of a collection.

The theory of operation of the *ft* command is simple. First, record the state of a given host's filesystems. Second, install the software in question on that

host using its documented installation procedure. Third, compare the before and after installation states of the host's filesystems and create a package that when deployed via SAFARI will result in the same changes to the host that the software's installation procedure made.

parcel

The *parcel* command is responsible for deployment of DSU's onto individual hosts. A special SAFARI configuration cluster is created for each host. The manager of a host cluster is the host manager. A host cluster defines the complete configuration of the host. It includes any files that uniquely identify the host (e.g., `/etc/hostname.le0` on Solaris) as well as a software configuration that specifies what DSU's are to be deployed onto the host.

The *parcel* command uses the list of DSU's to construct a virtual configuration of the host and then takes whatever steps are necessary to make the host comply with that configuration. *Parcel* supports a variety of mechanisms for abstractly selecting DSU's without having to precisely name each one. Examples include, selection by platform, by status, by revision number, by package family, and by image family. The goal of these mechanisms is to maximize the lifetime of a particular configuration specification. This means that once a host manager has expressed his intentions for the host, *parcel* will automatically keep that host synchronized with the DSU's in the repository.

For example, once a host manager has said "I want the latest revision of GNU Emacs that is certified as production quality," *parcel* will automatically make sure that when a new revision of GNU Emacs appears with the correct status, it will be deployed to the host, without the host manager having to say that he wants the new revision.

The host image from the host cluster is not really special in any way, other than it being the first image that *parcel* processes.

Parcel also offers the capability to choose whether DSU files to be deployed into host directories are copied locally or referenced in the repository via symbolic links. The ability to reference files in DSU's in the repository allows the host manager to effectively multiply the size of his local disk to accommodate host configurations that would normally require more disk space that is available. This can also be a very powerful tool for SU managers (who are typically host managers for their own desktop workstations). Deploying an **alpha** status DSU for testing via symbolic links is extremely fast, and reduces the develop-deploy-test cycle time. *Parcel* can even be instructed to deploy an unreleased DSU for pre-alpha testing by the SU manager.

Parcel is designed to be able to build a host from an effectively empty disk. For example, when we build Solaris hosts, we boot the host via the network,

format, partition, and mount the local disk, and then run *parcel* to install the software. Everything installed on the host comes from the repository. This means that recovery from catastrophic failure is the same as initial installation.

Parcel can be run as often as desired to update a host from its repository. Since extensive changes to the host may result, it is typically wise to run *parcel* only at shutdown or boot time. *Parcel* can also simulate an update so that the host manager can decide if an update is required and if he wishes to risk updating an operating host.

Cooperating Repositories

We have thus far engaged in considerable speculation regarding the practicality of multiple cooperating repositories. These speculations include:

- Demonstrating to vendors the value of a canonical distribution mechanism.
- Sharing of technical talent between repositories (e.g., operating systems are hard to put into a repository, but only one repository needs to contribute an operating system; then all other repositories can use it).
- Open source software can be packaged once by its developers or delegated builders and everyone can choose to use it or not based on their trust of the developers or builders.
- Local repositories can be used to cache DSU's from around the world, making a network of high performance deployment servers.

We don't propose that SAFARI is the canonical deployment vehicle. We only seek to show the benefits from having such a world-wide mechanism.

Availability

The CERT Coordination Center offers a web site, <http://www.cert.org/safari>, that describes the project in more detail and provides access to the available software components.

Future Work

Future work includes:

- Canonical seekable transportable format for packages and images.
- Managing dependencies between and among collections, packages, clusters, and images.
- Better cryptographic and message digest technology.
- General purpose configuration file delta processor, a la patch [14].
- Local file system access control lists support.
- Release conformance validation (structure and protections).
- Client-server administration.
- Taking snapshots of build and OBJ areas at package/image release time.
- Support for multiple collaborating repositories.

- Support for OSF's DCE/DFS.
- Supporting Windows NT 5 and CIFS, both as a server, to house a repository, and as a client.

Acknowledgments

SAFARI is built upon ideas that emerged from eight years of work at Carnegie Mellon University and the University of Pittsburgh. Between those two institutions, over 1,000 hosts are being managed by the precursors of SAFARI. The number of people who have significantly contributed to this work, directly and indirectly, is simply too great to enumerate here.

Author Information

Bill Fithen is a Senior Member of the Technical Staff at the Software Engineering Institute, currently working in the Networked Survivable Systems program which operates the CERT Coordination Center. Bill is the principal architect for several development projects, including SAFARI. Bill previously worked as the Manager of the University Data Network at the University of Pittsburgh. Bill earned a BS in Applied Physics and a MS in Computer Science, both from Louisiana Tech University. Reach him at <wlf@cert.org>.

Steve Kalinowski is a Member of the Technical Staff at the Software Engineering Institute, currently working in the Networked Survivable Systems program which operates the CERT Coordination Center. Steve is the leader of the team providing computing infrastructure services within the program. Steve previously worked on distributed computing environments as the UNIX Services Coordinator at the University of Pittsburgh, on the Electra electronics troubleshooting product at Applied Diagnostics, and on computer-integrated manufacturing systems at Cimflex Teknowledge, all in Pittsburgh, Pennsylvania, USA. Steve earned a BS in Computer Science from the University of Pittsburgh. Reach him at <ski@cert.org>.

Jeff Carpenter is a Member of the Technical Staff at the Software Engineering Institute, currently working in the Networked Survivable Systems program which operates the CERT Coordination Center. Jeff spends most of his time leading a team that provides technical assistance to Internet sites that have computer security issues or have experienced a security compromise. Before joining CERT, Jeff was a systems analyst at the University of Pittsburgh working in the computer center designing the university's distributed UNIX environment. Jeff earned a BS in Computer Science from the University of Pittsburgh. Reach him at <jjc@cert.org>.

Jed Pickel is a Member of the Technical Staff at the Software Engineering Institute, currently working in the Networked Survivable Systems program which operates the CERT Coordination Center. Jed is a member of the team that provides technical assistance to

Internet sites that have experienced a security compromise and is also a member of the team that coordinates responses, including CERT Advisories, to vulnerability reports. Jed earned a BS in Electrical Engineering from the University of California, San Diego. Reach him at <jpickel@cert.org>.

References

- [1] Sun Microsystems, Inc. 1997. *Application Packaging Developer's Guide* [online]. <http://docs.sun.com/ab2/coll.45.4/PACKINSTALL/@Ab2TocView>.
- [2] UNIX Systems Laboratories, Inc. 1990. *UNIX system V release 4 Programmer's Guide: System Services and Application Packaging Tools*. Englewood Cliffs, NJ: UNIX Press, Prentice-Hall.
- [3] Bailey, Edward. 1997. *Maximum RPM* [online]. <http://www.rpm.org/maximum-rpm.ps.gz>.
- [4] IBM. 1998. *AIX version 4.3 general programming concept: packaging software for installation* [online]. http://www.rs6000.ibm.com/doc_link/en_US/a_doc_lib/aixprgdd/genprog/pkging_sw4_install.htm.
- [5] Colyer, W. and Wong, W. 1992. *Depot: A Tool For Managing Software environments* [online]. <http://andrew2.andrew.cmu.edu/depot/depot-lisaVI-paper.html>.
- [6] Fithen, B. 1995. *The Image/Collection Environment For Distributed Multi-platform Software Development and System Configuration Management: A Tool for Managing Software Environments* [online]. <http://www.pitt.edu/HOME/Org/CIS-SN/SDR/public/ICE/index.html>.
- [7] Mui, L. and Pearce, E. 1993. *Volume 8 - X Window System Administrator's Guide*. Cambridge, MA: O'Reilly & Associates, pp. 196-197.
- [8] Transarc Corporation. 1998. *The AFS file System in Distributed Computing Environments* [online]. <http://www.transarc.com/dfs/public/www/htdocs/.hosts/external/Product/EFS/AFS/afsoverview.html>.
- [9] Seagate Software. 1998. *WinINSTALL* [online]. <http://www.seagatesoftware.com/wininstall>.
- [10] Garfinkel, S. 1994. *PGP: Pretty Good Privacy*. Cambridge, MA: O'Reilly & Associates.
- [11] Rivest, R. 1992. *RFC 1321: The MD5 Message-digest Algorithm* [online]. <http://info.internet.isi.edu/in-notes/rfc/files/rfc1321.txt>.
- [12] Tichy, W. F. 1985. *RCS: A System for Version Control*. Software Practice and Experience. vol. 15, no. 7, pp. 637-654.
- [13] Silverberg, I. 1992. *Source File Management With SCCS*. Prentice-Hall ECS Professional.
- [14] Wall, L., et al. 1997. <ftp://prep.ai.mit.edu/pub/gnu/patch-2.5.tar.gz>.
- [15] Johnson, M. K. 1997. *Linux Information Sheet: Introduction to linux* [online]. <http://sunsite.unc.edu/LDP/HOWTO/INFO-SHEET-1.html>.

- [16] Colyer, W. and Wong, W. 1992. *Depot: A Tool for Managing Software Environments* [online]. <http://andrew2.andrew.cmu.edu/depot/depot-lisaVI-paper.html#HDR6>.
- [17] Internet Engineering Task Force & Braden, R., ed. 1989. *RFC 1123: Requirements for Internet Hosts – Application and Support* [online]. <http://info.internet.isi.edu/in-notes/rfc/files/rfc1123.txt>.
- [18] Josey, A., ed., 1997. *Go Solo 2: The Authorized Guide to Version 2 of the Single UNIX Specification*. Englewood Cliffs, NJ: UNIX Press, Prentice-Hall, p. 525.
- [19] Free Software Foundation. 1996. <ftp://prep.ai.mit.edu/pub/gnu/autoconf-2.12.tar.gz>.
- [20] Lewine, D. 1991. *POSIX Programmer's Guide*. Cambridge, MA: O'Reilly and Associates. pp. 262-263.
- [21] Defert, P., et al. 1990. "Automated Management of an Heterogeneous Distributed Production Environment." *Proceedings of LISA*.
- [22] Manheimer, K., et al. 1990. "The Depot: A Framework for Sharing Software Installation Across Organizational and UNIX Platform Boundaries." *Proceedings of LISA*.

SSU: Extending SSH for Secure Root Administration

Christopher Thorpe – Yahoo!, Inc.

ABSTRACT

SSU[†], “Secure su,” is a mechanism that uses SSH [Ylonen] to provide the security for distributing access to privileged operations. Its features include both shell or per-command access, a password for each user that is distinct from the login password and easily changed, and high portability. By installing SSU, administrators build a solid infrastructure for using SSH for improving security in other areas, such as file distribution and revision control.

Introduction and Site Information

The EECS research computing environment at Harvard University is comprised of approximately two hundred workstations running eight variations of Unix. Users are primarily faculty, graduate students and researchers, most of whom need some level of root access to their workstations. Some students need only the ability to reboot their workstations or mount removable storage media, but many computer science researchers need full root access for their work. In addition, these Unix-savvy users ease the load on system administrators tremendously by performing administration tasks when possible.

Because the EECS environment is a research-oriented group, machines come and go on the network all the time, as do users who need privileged access. There are several independent research groups within EECS, and researchers in a group generally need privileged access only to that group’s machines. In addition, many researchers administer their own research machines while using our network and home directory NFS server. Since our environment is comprised of machines for which we provide various levels of administration, we built a system to give both administrators and researchers root access on various groups of machines. In addition, we made the system easy to install so that anyone setting up a new machine need only modify or create two files in addition to the ssh distribution (which everyone installs anyway.)

SSU sits on top of the widely-used secure shell protocol SSH. Because of its usefulness in providing security and ease of access to networked computing environments, ssh is now in use on all of the Unix-based machines within EECS. SSH can also be used for remote administration, where a trusted host uses the protocol to establish secure connections to other machines and execute privileged operations. SSH has

been successfully used to support the secure exchange of data for programs such as rdist [Cooper] and CVS [Cyclic].

In any large installation, key management is an important task for the proper installation and maintenance of SSH. Since we already used SSH, it was natural to extend the key management system we developed for remote root operations into a more complex system supporting SSU. By doing so, researchers adding a new system need only install SSH and add the trusted public identity key in root’s authorized_keys file. After this is done, SSU is installed automatically from the trusted host and keys are periodically updated. Note that SSH uses two types of RSA keys: one for host identification and another for user authorization. SSU does not use the host identification keys outside their normal use when ssh establishes a connection between two hosts. All SSU authentication uses 1024-bit RSA key pairs, completely separate from a host’s identity keys.

The features we required:

- The ability to easily redefine users’ root access
- Definition of access in terms of groups of machines and automatic update of machines’ root access configurations when these groups change
- The ability to administer machines without possessing a local user id
- Easy installation for researchers installing new systems
- Seamless portability between all of our operating systems

Features we wanted:

- A password for privileged operations distinct from the user’s login password and consistent for all operations
- Authentication of the user within the EECS domain before allowing root access

Description of SSU

Table 1 shows the commands and files used by SSU. Examples 1, 2, and 3 show how to reboot the local machine, gain a remote shell, and grant access to

[†]The bulk of this work was completed while the author was employed as a student systems administrator while an undergraduate at Harvard University. SSU is currently used in Harvard’s EECS (Electrical Engineering and Computer Science) research environment.

a new user. Note that this program asks for multiple command/host pairs, so that it is possible to define a different set of commands for each group of hosts.

Overview of How SSU Works

When users are granted access they are given access to commands chosen from a list of SSU commands (checked in a configuration file for sanity).

A configuration file is created or updated by running `ssu-user`. This file may be modified by hand to change users' permissions in the future. The reason for using this file is that if a user is granted access to groups of servers (as defined in the `netgroup` file or the `GROUPS` directory) and those groups change the user's permissions are automatically adjusted to correspond to the changes in those groups the next time `process-ssu` is run.

An RSA key pair is generated for each command, and the passphrase is set to be the same for all commands for that particular user (in order to facilitate ease of remembering the passphrases.) The `ssu-passwd` command makes it easy for users to change all of their SSU passphrases at once.

A user executes a privileged command by typing "`ssu command`" or "`ssu command@host`." The `ssu`

script then determines the appropriate identity to use, connects to the SSU port on the remote machine or loopback interface as root, and `ssh` executes the command as root. See Appendix D for a description of how SSH authenticates a user. Note that SSU does not allow `.rhosts`, `.shosts` or password authentication, and disables TCP/IP, X, and agent forwarding by default.

Key Management

Key management was probably the most difficult problem in implementing this solution. Since the system's security is based on RSA public/private key cryptography, it is vital to correctly administer the keys. First, we introduce the idea of a "trusted master." This machine holds an RSA private key for the root user that is trusted by the root users on all other machines on our network. (It is possible, and probably wise, to have multiple trusted masters.)

By configuring all the machines in the network to trust the root identity on this host, it allows all authentication for user root access to take place on the trusted machine. Connections with root access can be established from there. This also has the useful side effect of creating a machine that can establish secure, privileged connections to all other machines for any purpose – e.g., `rdist`, network monitoring and backups.

Command/File	Description
<code>ssu</code>	Used to invoke a privileged operation locally or remotely.
<code>ssu-passwd</code>	Used to modify a user's RSA passphrases for all SSU commands.
<code>ssu-user</code>	Administrators' tool for creating or modifying SSU privileges.
<code>process-ssu</code>	Processes the configuration files, generates the <code>authorized_keys</code> files, and pushes the files to the hosts.
<code>SSU.pm</code>	A Perl 5 module that contains local configuration settings and library functions used by <code>ssu</code> commands.
<code>ssu_cmdgroups</code>	Definitions of convenient groups of SSU commands.
<code>ssu_usergroups</code>	Definitions of convenient groups of SSU users. These groups may work with or be instead of the system group file.
<code>ssu_hostgroups</code>	Definitions of convenient groups of hosts for SSU. These groups may work with or be instead of the system <code>netgroup</code> file.

Table 1: User Interface.

```
% ssu reboot
Enter passphrase for RSA key 'cat:tcsh@eeecs': [passphrase]
Shutdown at Sun Sep 20 14:59:19 1998.
shutdown: [pid 19268]
Connection to localhost closed.
```

Example 1: Rebooting the local machine.

```
% ssu tcsh@herbert
Enter passphrase for RSA key 'cat:reboot@eeecs': [passphrase]
herbert#
```

Example 2: Gaining a shell on a remote machine.

Alternatively, private keys may be pushed to all of the machines on the network or placed in an NFS-mounted directory (see Appendix C for security concerns). With the keys available everywhere, users can gain root access through the loopback port. This allows restricted login access to the trusted master as well as the ability to gain root access even if the trusted master is inaccessible.

The master's trusted private key should be carefully guarded. We do not protect it with a passphrase so automatic privileged operations can execute without administrator intervention at boot time. In Appendix B we describe a method of passphrase protection that requires minimal administrator intervention for automatic operations. Using this method, however, it is required to enter the passphrase for each manually executed remote operation, e.g., updating the remote keys.

It is also possible to use SSU without allowing remote root access; to do this the keys must be distributed as described and connections as root allowed only through the loopback port. Pushing keys to remote hosts is difficult without remote root access. One solution is to make the clients periodically query the host where keys are stored to obtain the most recent `authorized_keys` file and the appropriate private keys.

If remote root access is desired, the trusted master is used to distribute lists of authorized keys to each of the machines for which privileged access is desired. These lists are constructed for each machine separately from the configuration files described below. Each entry in the list contains a command, any special

options with which `ssh` will execute the command (e.g., `environment="SHELL=/bin/false"` to prevent shell escapes from `vi` or `less`), and the public key for the user-command pair generated by `ssu-user` or `process-ssu`.

Since the private keys of these user-command pairs are protected by passphrases, even if they are captured via NFS sniffing or user negligence they are still secure. Some administrators may wish to place them on the trusted server's local disk if root access is to be limited to connections coming from the trusted server. Since we allow trusted connections from the trusted server and from the loopback port, we place the keys in the users' NFS-mounted home directories for convenience. (See Appendix C for further discussion of the security issues here.) We also store the keys on the trusted server so that if our NFS server fails it doesn't disable all privileged access.

Each user requires a separate key for each command. Fortunately, most users need only execute a few commands. For example, using "adminmenu," each user can be given a specific list of privileges from a selection of common administrative operations. The list of allowed operations is set in the configuration for each user. This cuts down on the administration of individual commands.

Because these keys are normal SSH public/private key pairs, the `ssh-agent` can be used to store the passphrases for these keys. We discourage this use of `ssh-agent`, as it creates two minor security holes. First, if a user adds the key to the agent, anyone can sit down at the computer later and execute the privileged commands without a passphrase. Second, if the login

```
% ssu-user newbie
SSU New User Configuration for newbie (Nathan Ewbie)

Enter the SSU commands newbie should have access to, space separated:
[Blank exits.]
bash reboot

Enter the hosts on which these commands should be accessible:
[Blank returns to previous prompt.]
bach handel mozart

Enter the SSU commands newbie should have access to, space separated:
[Blank exits.]
vi-aliases

Enter the hosts on which these commands should be accessible:
[Blank returns to previous prompt.]
mailhost

Enter the SSU commands newbie should have access to, space separated:
[Blank exits.]
[return]

Please remember to run process-ssu after you are finished adding new users!
```

Example 3: Granting access to a new user.

is shared, compromised, or a login is left open on another machine, a malicious user can set environment variables to use the running ssh-agent to gain access without a passphrase.

Configuration

Within the `/usr/local/etc/SSU` directory, there are five important configuration files: `SSU.pm`, `ssu_hostgroups`, `ssu_usergroups`, `ssu_cmdgroups`, and `ssu_config`. In addition, a `COMMANDS` directory contains a file containing definitions for each command SSU will be used to execute as root and a `HOSTS` directory contains the `authorized_keys` file for each host in the installation. The `HOST_DEFAULTS` directory contains files that are prepended to the host definitions created by `process-ssu` so that special defaults can be created on a host-by-host basis. `SSU.pm` is a Perl 5 module that contains important local configuration details such as the location of files and how to obtain the domain name correctly.

The `ssu_hostgroups` file contains a list of keywords that map to groups of hosts or other groups. For example, the lines:

```
lab-linux: bach handel mozart brahms
linux: lab-linux alfie betty
```

define groups `lab-linux` (`bach`, `handel`, `mozart` and `brahms`) and `linux` (`bach`, `handel`, `mozart`, `brahms`, `alfie` and `betty`). A group must be defined before use as a subgroup. Obviously groups cannot be subgroups of each other, so this "before" rule does not limit groups' definitions. Such group definitions allow the administrator to grant a user permission to execute commands on one or more groups of hosts, e.g., `mail-hosts`. When `mailhosts` changes, users with privileges for `mailhosts` automatically have their keys distributed to the new `mailhosts` map on the next key distribution update. Included with the SSU distribution is a utility that will generate `ssu_hostgroups` lines from an NIS [Sun] `netgroup` format.

The `ssu_cmdgroups` file contains keywords that map to lists of commands that a user in that command group may execute. The file behaves exactly like `ssu_hostgroups`, and command groups may contain other groups already defined. For example, if `helpdesk` staff needed a certain group of commands, an administrator might add the line:

```
helpdesk: passwd, lprm, reboot
```

The `ssu_usergroups` file contains keywords that map to lists of users. Again, groups may consist of usernames or other groups. Note that SSU reads the `/etc/group` file before processing this file, so groups defined there need not be redefined, and groups in `ssu_usergroups` may use groups defined in `/etc/group`. If a group is redefined in `ssu_usergroups` after being initially defined in `/etc/group`, a warning is printed and the old definition is lost. The `COMMANDS` directory contains several short files that function as SSU

command aliases. These aliases are used in the `ssu-user` script so that the full pathname and environment need not be specified for commands. This file contains part of line that will go into the `~root/.ssh/authorized_keys` file. Example 4 shows how one might configure files for a root `tcsh` shell and a command to modify the mail aliases. In Example 5 we set the `SHELL` environment variable to `/bin/false` to prevent the user from performing shell operations through `vi`. We set the path explicitly as well for security, and because `kill` is not in the same location on all of our hosts.

```
command="/usr/local/bin/tcsh"
```

Example 4: Configuring a root `tcsh`; file `tcsh`.

```
no-port-forwarding,no-X11-forwarding,
o-agent-forwarding,
environment="SHELL=/bin/false",
PATH="/bin:/usr/local/sbin:/usr/bin"
command"vi /etc/sendmail.cf;
kill -HUP 'cat /var/run/sendmail.pid'"
```

Example 5: Preventing shell operations from `vi`, file `vi-aliases`.

The `HOSTS` directory contains a file for each host on which SSU is to be run. It is initialized from the `HOST_DEFAULTS` directory each time `process-ssu` runs. `process-ssu` then appends to or creates files in this directory to complete its list of `authorized_keys` files. To do so, it examines the list of commands for each user in the configuration file, and creates a line in the appropriate host file for each user-command pair. These lines are constructed by concatenating the command definitions with the user's public key for that command and eliminating all internal newlines.

Once the user/command pairs have been processed, `process-ssu` then goes through each host in the `HOSTS` directory and sends the new `authorized_keys` file to the host via `scp` (part of `ssh`). If a passphrase is needed, the administrator should run `ssh-agent` before `process-ssu`.

Host Configuration

For security reasons, we want to limit privileged connections to the loopback port and trusted hosts. Since `ssh` does not allow limiting connections on a per-user basis, we run two `ssh` daemons – one on the standard port 22 for general access, and another on another privileged port (we use 122). The only recommendation we make with regard to the standard `ssh` configuration is that it contain the line `PermitRootLogin no` so that root access is only available through the special `ssh` daemon.

We then configure another `ssh` daemon to run for supplying privileged access, using a file including the following lines:

```
Port 122
RandomSeed /etc/ssh_root_random_seed
```

```

PidFile /var/run/sshd_root.pid
PermitRootLogin nopwd
RhostsAuthentication no
RhostsRSAAuthentication no
RSAAuthentication yes
PasswordAuthentication no
AllowHosts 127.0.0.1
            140.247.60.30 140.247.60.20

```

(See the source for a complete listing of the file we use.)

The Port, RandomSeed and PidFile lines prevent conflict with the standard ssh daemon. We turn password authentication completely off so that users will never be prompted for root's password – even if they know it. We allow only RSA authentication using the trusted keys and those generated from SSU-user. Finally, we restrict the hosts that may connect as root to the two trusted servers and the loopback port. This makes certain that even if intruders break into a non-trusted machine and gain a passphrase and private key, they cannot gain root access elsewhere on the network.

Related Solutions

Others have implemented similar solutions to this problem. These include `priv` [Hill], `sudo` [Courtesan], and `op` [Christiansen] which are very similar programs. These allow users to execute certain commands with root privileges. Each of these uses a configuration file describing privileged commands, users allowed to execute those commands, and the hosts on which and arguments with which the commands may be executed. Users invoke privileged commands by using a prefix (“`priv`,” “`sudo`” or “`op`”), then the command which is to be executed. While these systems are useful, we found them to be inadequate for our needs.

By extending our installation of SSH to SSU, we did not introduce any new binaries. All of the features SSU provides are written in Perl. SSH and Perl are already ported and thoroughly tested on all of the operating systems we use, and are utilities that we already use and maintain for other purposes. By using only existing binaries, the system is completely portable, and we avoid potential security holes introduced by writing and porting setuid C code. Given the highly diverse nature of our environment, portability and ease of installation is a very high priority.

None of the above solutions allows for a password distinct from login passwords in the password database. `Op` allows for specifying the password of another user; `priv` allows for that as well as “safe deposit box” authentication where two distinct users must type their password. These solutions are better, but login passwords are notorious for being sniffed and cracked, even in shadow password systems. `Priv` also provides mechanisms for password challenges and single-use passwords, but we concluded that these

solutions were too cumbersome for our users. SSU has four levels of security: the standard login password, the hosts' RSA key pairs, and the user's RSA keys and the passphrase protecting the private key.

SSU also is unique in that it allows privileged operations on a host to which the user does not have login access. If a researcher wishes to restrict user access or not to install NIS on a system, staff members might not have logins on systems (particularly those that are new or unstable). While one might argue that machines should always have staff logins, it is unrealistic to expect all researchers to repeatedly update their systems with the most up-to-date staff information. An SSU public key in root's `authorized_keys` file gives users the ability to perform operations on remote systems through authentication on a trusted host.

There are some useful features in these systems that `ssu` lacks. For example, `priv` allows the administrator to specify time of day and terminals at which users may execute privileged operations. SSH has no mechanism for supporting this, and hence, neither does SSU. With `sudo`, users need not retype their passwords each time `sudo` is executed – it “remembers” authentication for a few minutes. (As described above, we discourage SSU users from using `ssh-add` to add an SSU key to an `ssh-agent` for this purpose.)

Another advantage of `sudo`, `priv` and `op` is that they act more like “`su`” than SSU. Commands such as `sendmail` and `shutdown` “remember” the user id of the original user and report it, where SSU creates a login as the root user. While it might be possible to modify SSH or the login shell to solve this problem, that defeats the portability of our system. To do what we can, we do set the `USER` and `LOGNAME` variables of the new shell's environment to the root-invoking user when creating the `authorized_keys` files.

ksu

Before installing SSU, we used Kerberos [Neuman] `ksu` to give root shells to trusted users. While `ksu` requires a Kerberos root instance password distinct from the Kerberos password and the login password, it does not allow execution of limited privileged commands. The most significant reason we discarded `ksu` was that a full installation required compiling and maintaining Kerberos binaries on multiple platforms. Using Kerberos would have inconvenienced researchers, as they would have to install and configure it on each machine they added to the network. We found that SSH provided the security features we used Kerberos for; the administration cost of Kerberos for `ksu` alone outweighs its benefit.

s/key

`s/key` [Haller] and other one-time password schemes for granting root access do not provide much more administrative flexibility than giving users the

root password. Revoking access from a user who has a list of valid s/key passphrases involves distributing new passphrases to the other users who need them. These systems may provide extra security, but they do not solve the problem of managing access to privileged operations.

login

Sharing the root password with everyone is probably the simplest solution, but is also the least secure. When the root password changes, everyone who needs

to know it must be notified, and whenever anyone no longer should have root access, it must be changed. This solution is generally only viable in small operations with a small number of capable administrators.

Comparisons

Table 2 compares the various solutions.

Feature	root	s/key	priv	sudo	op	ksu	SSU
Same root-invoking password everywhere	•	-	1	•	1	•	•
Allows only specific commands	-	-	•	•	•	-	•
Each user has a unique password	-	•	•	•	•	•	•
Root passwords are independent from login passwords	-	•	2	-	-	•	•
Can gain root access locally if trusted master is unreachable	•	•	•	•	•	-	3
Requires authentication as a known user before gaining root access	4	-	•	•	•	•	•
The root password itself is never requested	-	-	•	•	•	•	•
Does not require a local user login	•	•	-	-	-	-	•
Simple to configure command limitations	•	•	•	-	-		
Can specify arguments to operations on the root-invoking command line	•	•	•	-	-		
Secure if user's login password is compromised	•	•	-	-	-	•	•
Secure if someone captures the local network traffic when run	5	•	•	•	•	•	•
Secure if root access is obtained through an unencrypted link	-	•	-	-	-	-	-

Key	
•	yes
-	no
space	not applicable
1	priv and op allow specified passwords for each command
2	priv allows for challenges and one-time passwords
3	SSU requires keys to be distributed or NFS-mounted for this to work. See Appendix C for a discussion of the security ramifications of this.
4	some operating systems restrict terminals where root can login and the users who may su
5	If a user telnets as root this is a risk; otherwise there is no problem.

Table 2: Comparison of various solutions.

Drawbacks

The most serious drawback of SSU is its reliance on the RSA keys. If the keys are kept on a master server or NFS server and that machine goes down, root access is disabled. The only way to completely remove this problem is by distributing the private keys to every machine's local disk. This (as does sharing them via NFS) increases the risk that an intruder could gain a private key. The solution with two master servers, each with local copies of the passphrase-encrypted keys, is probably solid enough for most installations.

Any SSU operation runs under a root login, whether executing a shell or a single command. Accountability is not provided on the local host, though the system logs from the ssh daemon provide ample information for this purpose. Broadcast messages and mail sent appear as from "root" rather than the invoking user.

SSU does not allow arguments to commands on the root-invoking command line. This forces the use of scripts or shell access for simple operations like renaming or killing processes, changing the ownership of files or setting the time for reboot of a machine. In addition, since the commands are executed in a remote manner, interactivity is limited. With `priv`, `sudo` and `op` a shutdown command is easily undone; SSU requires another connection to stop the shutdown process. Limiting command parameters is complicated in SSU, as it must be done using the backend script. However, there is less room for error in a commonly used language like perl than in a unique, single-purpose language.

Conclusion

SSU is a very useful tool for distributing root access among a large group of skilled researchers and faculty. Its most important features are that it allows distinct root passwords and configurable commands for each individual, uses a secure infrastructure for authorization throughout an installation, and is extremely portable. The security is solid and well-tested. It logs all commands or shells started for accountability. We have several examples of commands that can be used for common system administration tasks, and hope that others who use this system will share their work.

Availability

The SSU distribution is available via anonymous ftp at `ftp://eecs.harvard.edu/pub/cat/ssu` and contains the most recent version of this document, all of the programs referenced herein, and additional utilities and documentation as ssu is improved.

Acknowledgements

Special thanks to Michael Barrientos, who supplied Appendix D and provided moral support and excellent feedback during the preparation of this work.

Also special thanks to Peg Schafer, who encouraged me ("strongly" would be an understatement) to complete this work and submit it for publication at LISA. Phil Cox, for his help preparing the work for LISA.

Author Information

Christopher Thorpe graduated in June 1998 with an A. B. in Computer Science and Music from Harvard University. While at Harvard, Chris worked as a system administrator for Harvard's EECS (Electrical Engineering and Computer Science) research group. In addition to working as a system administrator, Chris was a head teaching fellow for introductory computer science courses at the university. Christopher is now employed at Yahoo! in Santa Clara, California as a member of the Yahoo! Store team. He lives in Sunnyvale, California and spends his negligible free time playing the piano and french horn, participating in community theatre, scuba diving, and playing computer games.

References

- [Christiansen] Christiansen, T. "Op: A Flexible Tool for Restricted Superuser Access," *Proceedings of the Workshop on Large Installation System Administration (LISA 88)*. Monterey, CA, USA, 1988. pp. 89-94.
- [Cooper] Cooper, M. "Overhauling Rdist for the 90's," *Proceedings of the Sixth Conference on Systems Administration (LISA 92)*. Long Beach, CA, USA, 1992. pp. 175-188.
- [Cyclic] CVS, Cyclic Software <http://www.cyclic.com>.
- [Courtesan] Sudo, Courtesan Consulting <http://www.courtesan.com/courtesan/products/sudo>.
- [Haller] Haller, N. M. "The S/KEY One-time Password System," *Proceedings of The Internet Society Symposium on Network and Distributed System Security*, 1994, pp. vi+173, 151-57.
- [Hill] Hill, B. University of California, Davis "Priv: Secure and Flexible Privileged Access Dissemination," *Proceedings of the Tenth USENIX System Administration Conference (LISA 96)*. Chicago, IL, USA, Sept. 29-Oct. 4, 1996. pp. 1-8.
- [Neuman] Neuman, B. C. and Ts'o, T. "Kerberos: An Authentication Service for Computer Networks," *IEEE Communications Magazine*, vol. 32, no. 9, pp. 33-38.
- [Sun] NIS+, Sun Microsystems Incorporated.
- [Ylonen] Ylonen, T. "SSH - Secure Login Connections over the Internet," *Proceedings of the Sixth USENIX UNIX Security Symposium*, San Jose CA, USA, July 22-25, 1996. pp. 214, 37-42.

Appendix A: United States Export Restrictions

The United States government restricts export of certain strong encryption algorithms. Since SSH is freely available from sites outside the US, we assume that anyone who wishes to install SSU can obtain and install a working version of SSH. The SSU distribution contains no encryption code; this is taken care of entirely by SSH. Those users who need information on installing SSH should consult the SSH FAQ (<http://www.uni-karlsruhe.de/~ig25/ssh-faq/>) and the SSH Distribution (<ftp://ftp.cs.hut.fi/pub/ssh/>).

Appendix B: Security Issues with Automatic Administration from a Trusted Host

The ssh-agent utility allows ssh users to enter the passphrases for a private key and saves the decrypted private key in memory. Further ssh connections use sockets administered by the ssh-agent. When a system is brought up, an ssh-agent running as root should be started with something like (assuming the trusted key is /root/.ssh/trusted)

```
csh% ssh-agent > /var/run/trusted-agent
csh% source /var/run/trusted-agent
Agent pid 1234;
csh% ssh-add -p
Need passphrase for /root/.ssh/trusted
(root@foo.com).
Enter passphrase: [passphrase]
Identity added: /root/.ssh/trusted
(root@foo.com).
```

(The -p option prevents ssh-add from starting an X11 window to read the passphrase if X is running.)

In scripts, one need only add the line

```
source /var/run/trusted-agent
```

before using ssh to make privileged connections to remote hosts. This will connect the process with the agent that already has the decrypted trusted key in memory. If any user other than root attempts to use this agent, they will fail.

This only increases security completely against someone obtaining the private key. The identity file is useless without the passphrase. If someone obtains a root shell on the trusted host while this is running, it is more difficult but still possible to gain remote access. In this scenario, they need only to examine /var/run/trusted-agent or use the agent-socket attack described below. Even if there is no trusted-agent file and a single ssh-agent is started for a master script running all remote operations, intruders could check the process table and the /tmp/ssh-root directories to find an ssh-agent process and agent-socket file with the same timestamp. They could then set the appropriate environment variables and "piggyback" on that agent to other hosts. Obviously this takes some knowledge of ssh and the local configuration, but is not particularly challenging. (Perhaps someone should add an option to ssh-agent to only allow connections from processes that are children of the agent itself.)

The other option, using the ~root/.shosts file, has its own weaknesses. (For some reason, SSH 1.2 does not like /etc/{s,}hosts.equiv for root.) The .shosts file essentially uses the hosts' private keys, which are never encrypted with a passphrase, as the sole source of authentication. With .shosts, if intruders obtain the trusted host's private key then they can spoof a connection. This may or may not be more difficult than gaining a shell or the root user's private key on the trusted host. Clearly, if they gain root access to the trusted host then they can ssh over through .shosts anyway. Since security-conscious configurations will limit trusted connections to trusted hosts, .shosts is no more secure than an RSA trusted key pair without passphrase protection.

With this in mind, if the trusted host is used to perform automated administration, configurations should ideally allow only trusted logins, and accept only ssh connections from other machines within the network. Such a trusted host should not run any other daemons that might compromise the security of the system, e.g., mail or telnet. In general, using any automated administration that trusts root on another host is a security risk, because when intruders obtain access to the trusted host, they can gain access remotely by modifying the automation scripts themselves.

Appendix C: Security Issues with Distributing Keys via NFS

While NFS is riddled with security holes, distributing the private RSA keys used for user authentication is not the security risk that it might seem to be, provided that these keys are encrypted with passphrases. (All of the scripts supplied with SSU disallow empty passphrases.) For this reason, even if an intruder were to obtain both pieces of a private/public key pair, it would still be necessary to obtain the passphrase. An intruder with both the public and private keys of an RSA key pair might be able to crack the passphrase if it were poorly chosen. Even if an intruder were able to crack the passphrase, however, it would then be necessary to gain a shell on a machine in order to use the passphrase to gain local root access. Malicious users with login access can often gain local root access and certainly degrade performance. For this reason, we believe that the barriers of obtaining the private key (which is never transmitted by ssh) and the public key, discovering the passphrase, and gaining login access to a machine are sufficient to prevent NFS-mounted SSU keys from becoming a security hole.

Another problem with distributing keys via NFS is that if the NFS server goes down, the keys are inaccessible. If the trusted master holds all of the SSU keys, then it is possible to obtain access to any machine through the trusted master.

Without using NFS, it is advisable to keep a local copy of the private keys on each host. The private

keys used for root access to a host should be pushed to it by the trusted master, or obtained by querying the trusted master periodically. In this way, an intruder with access must gain physical access to the files on the host in order to gain access to the private keys. This method allows local root access even if the trusted master and NFS server are inaccessible.

Appendix D: Establishing a Secure Connection via SSH

by Michael Barrientos and Christopher Thorpe

Server Resources		Client Resources	
<ul style="list-style-type: none"> * 1024 bit RSA private host key * 768 bit RSA server key, regenerated hourly and stored only in memory 		<ul style="list-style-type: none"> * Database of RSA public host keys 	
Steps			
SERVER		CLIENT	
	<-----	1. Establish a connection.	
2. Send the host and server public keys to the client.	=====>	3. Compare the host public key against a database of known hosts.	
6. Decode K with the private host and server keys, and ensure they are consistent.	<-----	4. Generate a 256-bit random number K.	
7. Offer client a choice of encryption algorithms to encrypt the data exchange using K as a key.	----->	5. Encrypt number K with the host and server public keys and send them to the server.	
9. Authenticate client user, trying in order, if allowed:	<-----	8. Choose an encryption algorithm (e.g., IDEA, 3DES, ...) and notify the server.	
* .rhosts/.shosts	<-----	Client's host key matches known key and encrypts random number.	
* hosts.equiv	<-----		
* .rhosts with RSA user identity challenge	<-----	Client encrypts random number with private identity key, returns result.	
* RSA challenge-response	<-----	Client provides login password. (Password is sent encrypted.)	
* password	<-----		
10. Port forwarding is set up for X, TCP/IP and ssh agent.			
12. If no command is listed in authorized_keys, give client a login shell or execute desired command. Otherwise, ignore request.	<-----	11. Client requests a shell/command.	
13. Standard input and output of the command executed go to and from socket linked to the client, encrypted with K.	<- 0-nn ->	Standard input and output of ssh process go to and from the socket linked to the server, encrypted with K.	

System Management With NetScript

Apratim Purakayastha and Ajay Mohindra – IBM T. J. Watson Research Center

ABSTRACT

Cost and complexity of managing client machines is a major concern for enterprises. This concern is compounded by emerging client machines that are mobile and diverse. To address this concern, management systems must be easy to configure and deploy, must handle asynchrony and disconnection for mobile clients, and must be customizable for diverse clients. In this paper, we first present NetScript, an environment for scripting with network components. We then propose a management system built with NetScript, where mobile scripts invoke components to perform management operations. We demonstrate that our approach results in a flexible, scalable management system that can support mobile and diverse client machines.

Introduction

Managing client machines in an enterprise is a challenging problem. Typical management operations include installing/updating applications, changing system files, monitoring performance, tracking client activities for diagnostics, and taking backups. The problem is compounded by certain characteristics of emerging clients. First, even within an enterprise, clients are heterogeneous – a mix of UNIX workstations, desktop PCs, laptops, palmtops, printers, and copiers. Management systems designed for traditional workstations are not readily applicable to laptops and palmtops. Second, emerging clients are often disconnected. Management systems therefore must support asynchronous and disconnected operations. For example, a management server should not fail, block, or have to actively retry a management task for a client that is disconnected (asynchronous server operation). On the other hand, a managed client should be able to prefetch and cache the right resources so that it is able to continue management operations when disconnected (disconnected client operation). Third, emerging clients are being used in more diverse application domains. Management systems therefore must support adequate customizability. In addition, management systems should themselves be easy to configure and deploy.

Current management systems and standards, such as Tivoli's TME-10 [10], Computer Associates' Unicenter [11], SNMP [13], CMIP [14], and DMI [15], are primarily designed for traditional workstations and desktops that are mostly connected and have enough local resources to host reasonably heavy-weight client-side management agents. They are hard to deploy and configure, they do not support asynchronous and disconnected operations, and they are not well-suited for a heterogeneous environment.

NetScript¹ is an environment for scripting with network components. In NetScript, a developer selects

required components from a distributed catalog and then writes a script invoking methods on these components as if the components are local. When a script is launched, the NetScript runtime dynamically determines component sites in the network and transparently migrates the script as needed. A remote component can also be transparently downloaded to a site where the script is currently executing.

The NetScript environment can be used in system management as follows:

1. Organize management code into appropriate components.
2. Use scripts to define management tasks that migrate to managed clients and dynamically locate and use the components at runtime.

The above approach offers a number of benefits. First, it extends the notion of scripting transparently to the network. Second, it centralizes management of components and scripts. Since management components can be downloaded at runtime, little management code needs to be pre-installed on managed clients, thereby drastically reducing configuration and deployment costs. Third, its script mobility naturally supports asynchrony and improves scalability by reducing load on the management server. Finally, its Java implementation improves portability.

The rest of the paper is organized as follows: the NetScript environment, illustrative examples of scripts and components, NetScript-based solutions to key system management problems, the technical challenges in using the NetScript environment for system management, related work in this area, and finally the conclusion. The appendices list a few scripts that are referred in the paper.

The NetScript Environment

A NetScript programmer writes a script by first selecting required interfaces from a distributed catalog and then invoking interface methods as if the components implementing the interfaces are local. An end user launches such a script into the network, where the NetScript runtime dynamically determines the

¹A prototype implementation of NetScript is freely available at <http://www.alphaworks.ibm.com/formula>.

component sites and transparently moves the state of the script to the component sites as necessary. It is also possible that the NetScript runtime downloads a component to the script execution site, or migrates both the script and the component to a third site.

The rest of this section summarizes different aspects of the NetScript environment including its component model, scripting language, and the runtime environment. This discussion is intended to provide only a reasonable background for the rest of the paper. Please refer to [2] for a more comprehensive discussion of NetScript².

The Component Model

The NetScript component model is based on well-known object-oriented programming concepts of interfaces, components, attributes, and globally unique identifiers. In NetScript, an interface is a group of semantically related methods or functions. A component is an implementation of the interface. A component can implement one or more interfaces. Each interface and component is identified by a globally unique identifier (GUID)³, called InterfaceID and ComponentID respectively. Interfaces and components may have attributes associated with them. Attributes may provide informative description (such as semantics and suggested use) or other parameters (such as manufacturer, usage cost) that may allow the NetScript runtime to select one component over another. Attributes are also identified by GUIDs called AttributeIDs.

Components in NetScript could be Java Beans or Java-wrapped native (e.g., ActiveX) components. Interfaces and components are advertised in a distributed catalog. Currently the catalog is implemented using the LDAP [3] distributed directory services. Interfaces and components are hierarchically organized in the directory. Currently only browsing function is supported whereby a programmer can browse the catalog and select required interfaces. Support for a more intelligent search function is planned.

The Scripting Language

The NetScript scripting language is an extension of the BASIC programming language. In addition to standard control constructs, the language has a few NetScript specific additions to create component instances and to make method invocations. For security reasons the language has no vocabulary for system operations such as direct memory access, file access, and network access. The language is deliberately kept simple for it to suffice as a glue language that glues components together.

For creating components, the language provides the `createComponent` keyword. The syntax is:

```
<varname>=createComponent(<interfaceName>,  
[<filter>]) [at <locationName>]
```

where `varName` is a handle to the resulting instance, `interfaceName` is the InterfaceID that the component should support, and `filter` is a boolean expression of desired attribute name-value pairs. Optionally, using the `at` keyword, one can also specify the `locationName` indicating where the component should be instantiated. Under the covers, the runtime contacts the component catalog to locate a component that implements the required interface and then either migrates the script to the component location or downloads the component to the script's location, instantiates the component, and stores the handle for the instance in the script's `varName` variable. The syntax for performing method invocations is as follows:

```
[<resultVar>=]  
<varName>.methodName(<arg1,...,argN>)
```

where `resultVar` is the variable to store the results of the invocation, `varName` is the handle for the component instance, `methodName` is the name of the interface method, and `<arg1,...,argN>` is the list of arguments.

We use a BASIC-like scripting language for reasons of free experimentation and convenience, including the fact that a free Java-based interpreter for it was available that we could modify and add our extensions easily. Use of a more standard scripting language like Tcl or Javascript is planned.

The Runtime

The NetScript runtime has been implemented in Java. The runtime has a script interpreter, an execution engine, and a set of shared services such as directory, instance management, and communication. The runtime also has built-in support for security and access control, garbage collection, monitoring, and failure detection and recovery. The NetScript environment also has command-line and web-based tools for the user to launch, monitor, and control executing scripts. Figure 1 gives a broad overview of the important parts of the NetScript runtime.

A script is first launched on a machine using one of the NetScript tools. The NetScript runtime initializes some data structures and starts executing the script. When executing a `createComponent` call the runtime may need to migrate the script to another runtime executing at a different host. Parts of the script including program counter, stack and data heap are transferred, while component instances that the script may have created locally are kept as part of the residual script state. Every runtime has a per-script instance manager that keeps track of component instances that the script may have created. On method invocations, the runtime queries the instance manager to decide if the script needs to migrate to the location of a component instance. On completion, the script typically returns to the home machine from where it was launched. The runtime at the home machine then

²NetScript was called NetPebbles at that time.

³A GUID is generated using a combination of hardware address, current time, and a random long integer.

initiates garbage collection to clean up instances and data structures that may have been created by the script on other hosts.

An Example

In this section we discuss a motivating example of how NetScript may be used to perform certain system management tasks. First, we discuss an example script by walking through its execution assuming certain components are available for the script to use. Next, we discuss how one may write one of the constituent components and incorporate the component in the NetScript environment to make everything work together.

An Example Script

Figure 2 shows an example script that discovers the version of an application (such as Lotus Notes) installed on every machine in a department and then displays the results on some specified machine. Figure 3 provides a visual analog for the execution of the

script that shows which parts of the script execute on which machines, how the script migrates, how the run-times interact with the component catalog and download components.

The script starts on an admin machine called "helix." The script first attempts to create a component that implements the interface "IDomainAdmin." The runtime uses the component catalog to locate the actual host for a component that implements the interface. When such a host is located, the script execution is suspended and the script is migrated to that host. The NetScript runtime on the destination host, which happens to be the component server in this case, instantiates the component and resumes script execution. The script invokes the "getMembers" method on the instance to get all member machines for "Department 931B." Then for every machine in the department, the script uses the "createComponent" function with the "at" clause to migrate to the machine as well as download the component that implements the

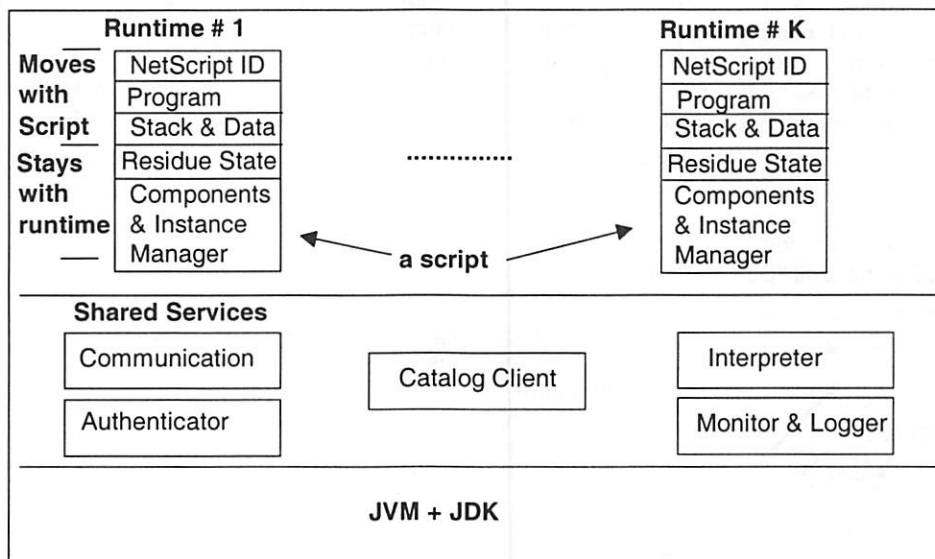


Figure 1: The NetScript runtime: scripts and shared services.

```
home = "helix.watson.ibm.com"
a = createComponent("IDomainAdmin")
locations = a.getMembers("Department 931B")
n = length(locations)
dim ver[n]
for (i=0; i < n; i=i + 1)
    b = createComponent("ISystemSniffer") at locations[i]
    ver[i] = b.getVersion("Lotus Notes")
endfor
c = createComponent("IDisplay") at home
c.showList(ver)
exit
```

Figure 2: A script for finding out Lotus Notes versions installed on department machines.

interface "ISystemSniffer." Like before, the runtime uses the directory service to locate a component host but instead of migrating to the component host, the component is downloaded from the component server to where the script migrates as a result of the "at" clause. The runtime at each machine instantiates the downloaded component and resumes the script execution. The script invokes the "getVersion" method on the instance to obtain the version of "Lotus Notes" and stores in a script array. Finally, the array is displayed using the "IDisplay" component at a specified machine. To improve performance, the NetScript environment also allows a user to "fan-out" a script simultaneously to a number of machines.

An Example Component

The components in this example can be written in pure Java, or can be Java-wrapped native code. The "IDomainAdmin" interface is one that manipulates user information. Conceivably the component implementing the interface can be written in pure Java that perhaps uses JDBC to perform actual database operations. Alternatively, it can also be implemented as a component that reads and updates simple files that contain user information. As long as the component supports the interface methods and semantics, its different implementations does not necessitate changes to

the existing scripts. The component implementing the "IDisplay" interface can also perhaps be a pure Java component that uses 'java.awt' methods for user interaction.

The component of interest however, is the one that implements the "ISystemSniffer" interface. In its generality, this interface will support not only methods that get version information for applications, but perhaps methods for sensing load conditions, memory availability, disk-space availability, battery power, etc. Since these operations are platform dependent, the component is likely to contain native code. In Appendix A we have outlined a simple implementation of the component that supports only the "getVersion" method for a few applications. Appendix A.1 lists java code that is only a scaffolding for the underlying native code in the C language. The Java Native Interface (JNI) is used to communicate across Java and C. Appendix A.2 lists the C code that implements the "getVersion" method for a Windows 95 or Windows NT platform. The C code simply traverses the windows "registry" and looks for specific key-value pairs to determine the version information. The native implementation will clearly differ if the same method is implemented for an AIX system that has different ways of storing application information.

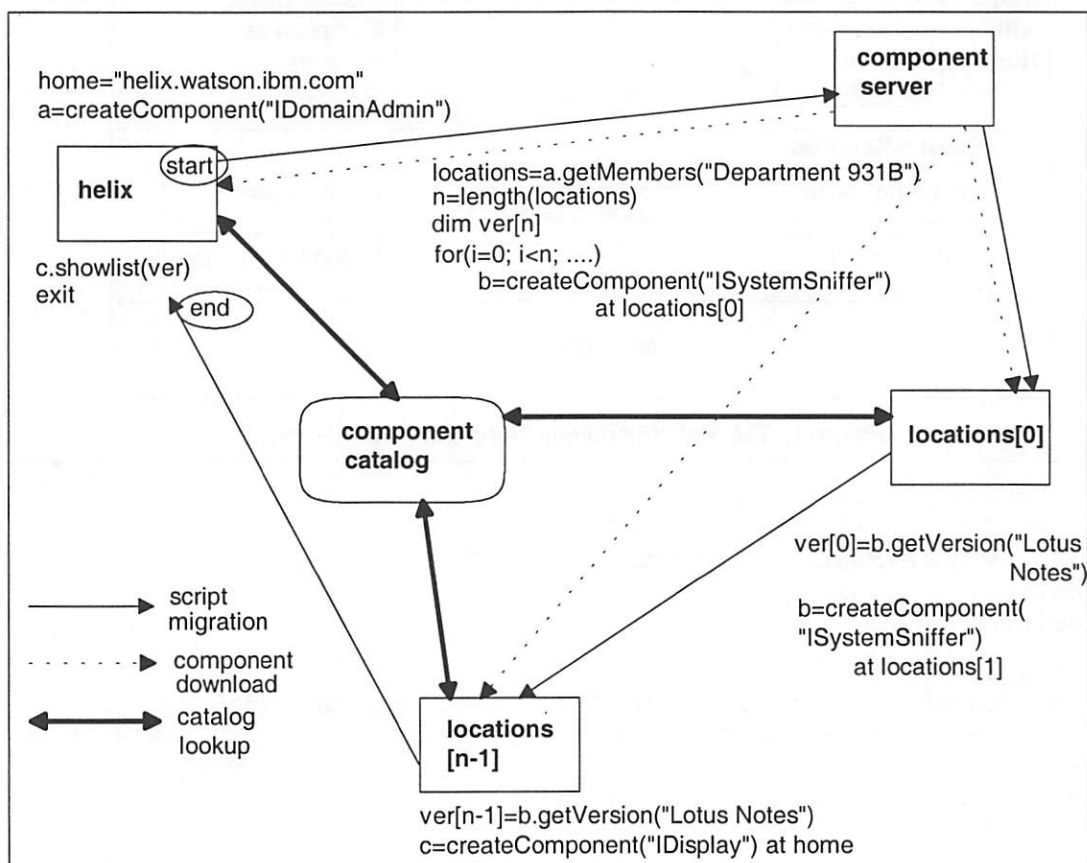


Figure 3: Visual depiction of the execution of the example script.

The classes and native libraries (if any) such as "dll"s for Windows systems and "so"s Unix systems are packaged as Jar files for distribution. The component is entered in the component catalog at its appropriate position. The component entry in the catalog specifies its location and the name of the Jar file in which the component is packaged. A NetScript runtime finds component location information from the catalog, unpacks the Jar file, and instantiates the component.

The above example embodies a simple programming paradigm where powerful components on the network are threaded together to perform an actual management operation. The NetScript environment itself does not attempt to dictate what system functions should be captured in reusable components and what should be captured in scripts. In system management, perhaps "functions" such as looking up a registry/database, copying/moving files, and checking process/memory status should be captured in components, while "decisions" or "control" such as identifying when memory is low, deciding that an update needs to be applied should be captured in scripts. Note that the intent of NetScript is not to write "core" management code but to help organize core management functions into components such that actual management tasks can be easily specified in simple scripts.

NetScript Solutions for System Management Challenges

Modern management systems need to be flexible. Changes in management infrastructure should not mandate a large-scale re-deployment on managed clients. In addition, the system should allow high levels of customizability for clients without the need to modify client-side code and hence forcing re-deployment. Modern management systems should also be able to contend with client disconnection and heterogeneity in managed clients or devices. In this section we will discuss how the NetScript approach can address these challenges.

Flexibility

Many management systems, including standards and products, rely on a management agent installed on a managed client. The management agent on the client is controlled by a management server. The main problem with this approach is that when a management agent is deployed, it becomes a fixed contract between the server and the managed client. Therefore any change in the behavior of the management system implies the client agent be changed and re-deployed. A second problem is customizability. Although some preprogrammed customizability can be added to the client agent, such an approach will be untenable as clients and users get more diverse.

NetScript solves this problem naturally and elegantly. All that needs to run on a managed client is the

NetScript runtime (only 40KB static footprint at this time). The runtime is generic and simple and is not specific to system management. It can also be used for executing the client's personal scripts, e.g., one that tracks stock prices for the user. The runtime is also capable of running as a servlet under a web server such that the runtime itself can be downloaded when appropriate. A NetScript-based management system closely approximates a zero-install client.

In NetScript, management components can be centrally stored and managed in some server. They can be changed freely without the need for deployment. The scripts can also be maintained centrally. The ability of the scripts to migrate to clients and download required components precludes the need for modifications at the client.

Appendices B.1 and B.2 illustrate how flexible and customizable a NetScript-based management system is. Appendix B.1 shows a script similar to the one in section 2. This script updates an application on all machines in a department⁴. There could be times when some clients will not want this update. For example, the reason could be that the client is connected over a slow phone line, or the client is making an important presentation. To incorporate such customizability into the system, the system administrator only needs to write a "Update Policy" component (which can be made arbitrarily powerful) and then change a few lines in the script (shown in italics in Appendix B.2). Note that no changes or deployment at the client is necessary to incorporate this customization.

From these examples one can infer that the basic philosophy of componentizing management code and threading those components at runtime via a NetScript script results in a management system that is immensely flexible and customizable. In addition, the NetScript approach results in better scalability because "control" migrates with the script. The central management server is not overloaded trying to coordinate activities of various client agents.

Asynchrony and Disconnected Operations

Users in modern enterprises are increasingly using laptops and other mobile devices that are frequently disconnected. Modern management systems must therefore contend with disconnection. First, servers should be able to schedule or launch management tasks anytime independent of whether clients are connected or disconnected. Without an explicit attempt-and-retry, the management system should be able to implicitly propagate appropriate tasks when clients reconnect (asynchrony). Second, the clients may choose to disconnect when a management

⁴For simplicity, the scripts show machine updates in a serial fashion. In practice that will not work well. NetScript environment includes support to launch scripts in parallel to various machines.

operation is in progress. The runtime on the client should be prepared to handle such disconnections and should prefetch appropriate components to allow the management operation to continue off-line (disconnected operation).

The NetScript runtime has implicit support for handling network disconnections. When a running script needs to migrate to another location, the runtime periodically attempts to transfer the script in configurable intervals until it succeeds or exhausts a configurable timeout.

Supporting disconnected operations in NetScript requires prefetching of appropriate classes in preparation for disconnection. As a first attempt we are considering supplying annotations in the script to help the runtime prefetch the right components. This approach is more practical than the runtime trying to be intelligent about the semantics of a script. We are also considering support for persistent scripts that are saved when a client shuts down and are revived when the client restarts. Support for disconnected operations and persistence is being designed and has not been implemented yet.

Heterogeneity

With the increasing popularity of networked devices it is reasonable to assume that in the near future managed systems will not only include traditional workstations and desktop PCs but also include devices such as laptops, palmtops, printers, and copiers. The managed system therefore needs to support heterogeneous and diverse set of clients. They will have different *modus operandi*, different operating systems (some of them will not even have one), and different functional roles. In some, application management will be important (laptops), in others function and capability management will be important (printers), and in yet others pure data management will be important (palmtops). If a management system even wants to begin to address this wide range, it has to be lightweight, portable, and adaptable.

NetScript is reasonably lightweight (40KB footprint). Since it is downloadable, it can also free up valuable space for more important applications in space constrained palmtops. It is portable to any Java-capable platform (may also be packaged with a lightweight Java Runtime Environment). It is also adaptable to availability of resources such as the network.

Consider printers as an example of a new type of managed client in a comprehensive management system.

Printers have errors such as toner low, paper out, paper jam, memory overflow, bad configuration, etc. Today we discover those errors only when we go to pick up a printout, even though the printer firmware has already detected and reported the error on its console. In the near future it is reasonable to expect for

printers to have IP addresses and be Java-enabled. With a NetScript-based management system, a server can send down a monitoring script to the printer, which basically snoops on the printer, and upon error, migrates to notify appropriate parties. Appendix C.1 shows a NetScript script that may be used for this purpose. The script uses the "IPrinterManager" component to obtain the printer's location, then migrates to the printer and downloads the "IPrinterAssist" component that is able to access the printer firmware. The script then waits in a loop checking for error conditions. When an error is detected, it first tries to fix the error if possible, or else it notifies the administrator. One can also imagine scripts that can visit a number of printers and enqueue a specific job in a printer that has perhaps the shortest queue length or the smallest total size for all jobs in the queue.

Issues in using the NetScript Environment for System Management

The practical use of NetScript in a real management system depends on a number of factors such as acceptability of its programming model including the scripting language and the component model, security, reliability, and monitoring and debugging support. This section discusses these issues in the NetScript environment.

The Scripting Language Dilemma

The NetScript scripting language is an extension of BASIC. It is purposely kept quite simple such that scripts are easy to write and modify. It is however, unfamiliar and not as prevalently used such as Tcl [4], Perl [5], or Javascript [6], therefore its acceptance in the system management may be in question. Scripting languages are usually extensible and therefore it should not be difficult to incorporate NetScript specific extensions (such as `createComponent`) into other languages. Why then, are we not using Tcl extensions? The reason is that scripting languages such as Tcl are quite powerful and can directly access files and sockets. Such capabilities might be undesirable (for reasons of security and simplicity) in a mobile script such as NetScript. We may consider using a proper subset of Tcl (such as `safe-Tcl` [7], with NetScript extensions) in the future. Since NetScript is currently under active experimentation, we used a language for which we found a public domain Java-based interpreter that we could easily modify to suit our needs.

The "Non-Standard" Component Model

The NetScript component model borrows from the Java Beans model as well as the ActiveX [8] model but is also distinguishable from both. The properties and interfaces supported by a bean cannot be ascertained unless the bean is instantiated and introspected. We believe that a system manager needs to know about the functional characteristics of a component before instantiating it. Therefore, like the

ActiveX registry we use a component catalog (implemented on LDAP distributed directories). However, in ActiveX, the programming model is component centric because a component is first located in the registry then it is ascertained what interfaces it implements. In NetScript the programming model is interface-centric. When writing scripts, the system manager simply chooses some functionality (syntactically and semantically characterized by an interface) from a catalog and the runtime locates that functionality in the form of a component.

Security

With mobile code such as in NetScript there is always a security concern. A malicious "management" script can harm a client. The users of the scripts must be authenticated and authorized to execute on certain machines or use certain components. NetScript provides mechanisms for attaching a "principal" with a script. Upon receipt of a script, the NetScript runtime authenticates the principal and verifies that the principal has execution rights on the local host. When a script accesses a component, the principal attached to the script is provided to the component catalog. Component entries in the catalog list principals that are allowed to use the component. Only if the provided principal is included in the list, a component location is returned to the requesting runtime. We have implemented a DCE-based [9] authentication and access control mechanism with NetScript that is suitable for intranet deployment.

Security issues however, reach further than simple authentication and access control. A NetScript runtime should prevent two different NetScripts from interfering with each other (isolation). The runtime uses per-script instance managers and class loaders to implement isolation. The NetScript environment also does not allow for editing a running script because of possible security breaches.

Reliability

Reliability is also a concern with mobile code. When a management script dies somewhere how can the system manager find out where it died and what actually happened? For scripts that are long-running (e.g., one that monitors network load on a router), what happens if the system is re-booted? How does one ensure that all the component instances of a failed script are garbage collected? Is it possible to recover a script after the host on which the script was executing crashes?

To survive across machine reboots, we have implemented mechanisms whereby a NetScript runtime responds to the "shutdown signal" (available in most operating systems) by saving its internal data structures and scripts in persistent storage and recovering from persistent storage at system startup. We have also implemented a version of the NetScript runtime that uses a reliable and non-blocking application-to-application transfer protocol called MQSeries [22].

The use of a reliable and non-blocking transport mechanism has allowed us to design simple protocols for reliably tracking a NetScript. We have also implemented limited checkpointing capabilities that allows some scripts to be recovered after a system crash.

Monitoring and Debugging

To be successful as an extensible, programmable environment, NetScript must include reasonable support for monitoring the execution of scripts and debugging scripts. Tightly controlling the execution of mobile code is a difficult problem. The NetScript environment, however, provides support for locating a script, retracting a script from any location, or killing a script. Scripts are identified by GUIDs generated and assigned to a script at the start of its execution. Any errors that are encountered by a script, including exceptions generated by components, are reported to the user when the user requests for the status of a script that has failed. Features like break-points, single-step execution are being considered but are not currently implemented.

Related Work

NetScript and Other Management Systems

Architecturally, most management systems such as Tivoli's TME-10 [10], Computer Associates' Uni-center [11], Marimba [12], IBM TJ.Watson Research's SysCtl [23], and Igor [24], rely on a installed client agent that works under commands from a central server in a request-response fashion. Network management standards such as SNMP [13] and CMIP [14] also imply the same architecture, and so do desktop management standards such as DMI [15]. For reasons cited in Section 2 this approach is fundamentally not flexible and scalable. Enhanced functionality often results in making the client side agent even more complex, thereby making it harder to maintain, re-deploy, and configure. None of the above systems also naturally support asynchronous operations although one can imagine retrofitting such function to them. None of the above systems (including implementations of the standards mentioned) are portable across operating systems, let alone different device classes such as desktops, laptops, palmtops, and network devices.

Management by Delegation (MbD) [16] partially addresses the scalability and flexibility limitations of SNMP-style systems. MbD proposes an architecture whereby delegation agents could be sent down from servers to clients that could then invoke delegation procedures stored on clients. MbD however, is not quite as flexible as NetScript because the delegation procedures themselves are not downloadable. The programming model is also not based on scripting, which is popular in the system administration community.

NetScript and Other Infrastructure Technologies

One may argue that given suitable management components one can build a similar management system using technologies like Java/RMI [17], CORBA

[18], or DCOM [8], or mobile agent technologies like IBM Aglets [19], or Agent Tcl [20], or Telescript [21], why use NetScript?

With Java/RMI technologies, per-component "ImplServers" have to be running on component hosts. This can cause difficulty in deployment. Moreover, for long running methods (such as monitoring a printer) RMI-like technologies will need to maintain a long running connection. Remote polling using short lived connections is not scalable.

Functionally a mobile agent technology such as IBM Aglets can do whatever NetScript can do. However, the script and component based programming model in NetScript is appreciably simpler. Same functionality is substantially easier to code and deploy using NetScript (see [2]).

Conclusions

In this paper we have summarized a technology for scripting with network components and argued for its gainful use in system management. Our prototypes have shown that using the NetScript technology in managing systems can overcome problems of flexibility/customizability, client mobility, asynchrony, and disconnection, and heterogeneity. At the same time we believe that NetScript provides an attractive and simple programming model for the system administration community.

For a fully functioning management system, components that do the actual work still have to be written. Some components can be purely Java and hence usable in all Java-enabled managed platforms. Some components will have to use native code for a long time to come (e.g., registry access component for Windows). These pieces of code have to be written in any management infrastructure that wants to perform the same functions. NetScript's contribution is proposing an elegant way to program with these components as available, granted pieces of function. As a result, the management infrastructure built around those components becomes more flexible, scalable, customizable, portable, and above all, "modern."

Author Information

Apratim Purakayastha received his Ph.D. in Computer Science from Duke University, where he was awarded a graduate fellowship in 1992. He worked in the parallel file systems area for his dissertation. Purakayastha joined IBM upon graduating in 1996. At IBM, he has worked on building Java-based technologies such as Thin-Client Application Framework and NetScript. He belongs to several professional organizations, including Usenix and ACM. Reach him at apu@us.ibm.com.

Ajay Mohindra received his Ph.D. in Computer Science from Georgia Institute of Technology in Atlanta. While earning his degree, he participated in the design and implementation of a distributed object-

based operating system. Mohindra is a member of IEEE (Institute of Electrical and Electronics Engineers), ACM, and Usenix. Reach him at ajaym@us.ibm.com.

References

- [1] The Java Beans home page. <http://java.sun.com/beans>.
- [2] Ajay Mohindra, Apratim Purakayastha, Deborra Zukowski, Murthy Devarakonda. "Programming NetWork Components Using NetPebbles: An Early Report." In *Proceedings of the Fourth Annual Usenix Conference on Object Oriented Technologies and Systems (COOTS)* April, 1998.
- [3] Timothy Howes and Mark Smith. "A Scaleable Deployable Directory Service for the Internet." In *Proceedings of INET 95*, 1995.
- [4] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley Publishing Company, 1994.
- [5] Larry Wall and Randall Schwartz. *Programming Perl*. O'Reilly and Associates, Inc. 1994.
- [6] *The JavaScript Guide*. <http://developer.netscape.com/docs/manuals/communicator/jsguide4/index.htm>.
- [7] *Safe-Tcl*. <http://sunscript.sun.com/plugin/safetcl.html>.
- [8] David Chappell. *Understanding ActiveX and OLE*. Microsoft Press, 1996.
- [9] Charles Knouse. *Practical DCE Programming*. Prentice Hall, 1995.
- [10] Rolf Lendenmann, Jennifer Nelson, Janet Selby, Carlos Patino Lara. *An Introduction to Tivoli's TME 10*. Prentice Hall, 1998.
- [11] Computer Associates Unicenter. <http://www.cai.com/products/uctr.htm>.
- [12] Marimba, *How Software Goes Down to Business*. <http://www.marimba.com>.
- [13] William Stallings. *SNMP, SNMP v2, and CMIP*. Addison-Wesley, 1993.
- [14] William Stallings. *Network Management*. IEEE Computer Society Press, 1993.
- [15] *Desktop Management Interface*. <http://www.dmtf.org/tech/specs.html>.
- [16] German Goldszmidt and Yechiam Yemini. "Distributed Management by Delegation." In *Proceedings of the 15th International Conference on Distributed Computing Systems*, 1995.
- [17] Ann Wollrath, Roger Riggs, Jim Waldo. "A Distributed Object Model for the Java System." *Proceedings of the 2nd Conference on Object Oriented Technologies and Systems (COOTS)*, 1996.
- [18] *CORBA2.0/IIOP Specification*. <http://www.omg.org/corba/c2indx.htm>.
- [19] Danny Lange and Daniel T. Chang. *IBM Aglets Workbench, Programming Mobile Agents in Java*. <http://aglets.trl.ibm.co.jp/whitepaper.htm>.

- [20] Robert S. Gray. "Agent Tcl: A Transportable Agent System." In *Proceedings of the Workshop on Intelligent Information Agents*, in the *Fourth International Conference on Information and Knowledge Management*, December 1995.
- [21] *Telescript Technology: The Foundation for the Electronic Marketplace*. <http://www.genmagic.com/Telescript/Whitepapers/wpl/whitepaper1.htm>, 1996.
- [22] *MQSeries*. <http://www.software.ibm.com/qseries>.
- [23] Salvatore DeSimone and Christine Lombardi. "Sysctl: A Distributed System Control Package." In the *Proceedings of the 7th Usenix LISA Conference*, pages 131-143, November, 1993.
- [24] Clinton Pierce. The Igor System Administration Tool." In the *Proceedings of the 10th Usenix LISA Conference*, pages 9-18, September, 1996.

Appendix A: "SystemSniffer" and Example Component

Appendix A.1: SystemSniffer.java

```
package COM.ibm.netpebbles.components.systemsniffer;

public class SystemSniffer{
    public native String getVersion(String appname);
    static {
        System.loadLibrary("COM.ibm.netpebbles.components.systemsniffer.vernative");
    }
}
```

Appendix A.2: vernative.c

```
#include "COM_ibm_netpebbles_components_systemsniffer_SystemSniffer.h"
#include <windows.h>
#include <string.h>
#include <mbstring.h>
#include <stdlib.h>
#include <stdio.h>

JNIEXPORT jstring JNICALL Java_COM_ibm_netpebbles_components_
    systemsniffer_SystemSniffer_getVersion(JNIEnv*
        env, jobject obj, jstring appstr){

    HKEY key;
    LONG retcode;
    char keyname[50], class[50];
    DWORD keynamesize = 50, classsize = 50;
    FILETIME lastwrittento;
    int i;
    char str[50];
    char ver[50];
    DWORD valuetype;
    DWORD version = 50;
    const char *appname;

    appname = (*env)->GetStringUTFChars(env, appstr, 0);

    if(!strcmp(appname, "Lotus Notes")) {
        if( RegOpenKeyEx( HKEY_LOCAL_MACHINE,
            "SOFTWARE\\Lotus\\Notes", 0, KEY_READ, &key) == ERROR_SUCCESS){
            for(i=0, retcode=ERROR_SUCCESS; retcode==ERROR_SUCCESS; i++){
                if((retcode = RegEnumKeyEx(key, i, keyname, &keynamesize, NULL,
                    class, &classsize, &lastwrittento)) == ERROR_SUCCESS){
                    str[0] = '\0';
                    strcat(str, "SOFTWARE\\Lotus\\Notes\\");
                    strcat(str, keyname);
                    if( RegOpenKeyEx( HKEY_LOCAL_MACHINE, str, 0, KEY_READ, &key) ==
                        ERROR_SUCCESS){
                        if(RegQueryValueEx(key, "Version", NULL, &valuetype, ver, &version)
                            == ERROR_SUCCESS){
```

```

        printf("Found version %s\n",ver);
        RegCloseKey(key);
        return((*env)->NewStringUTF(env,ver));
    }
}
}
return((*env)->NewStringUTF(env,"Installed but version unavailable"));
}
else {
    return((*env)->NewStringUTF(env,"Not Installed"));
}
}

if(!strcmp(appname,"Netscape Navigator")) {
    if( RegOpenKeyEx( HKEY_LOCAL_MACHINE,
        "SOFTWARE\\Netscape\\Netscape Navigator",0,KEY_READ,&key) ==
        ERROR_SUCCESS) {
        if(RegQueryValueEx(key,"CurrentVersion",NULL,&valuetype,ver,&versize)
            == ERROR_SUCCESS){
            printf("Found version %s\n",ver);
            RegCloseKey(key);
            return((*env)->NewStringUTF(env,ver));
        }
        return((*env)->NewStringUTF(env,
            "Installed but version < 4. registry not properly configured"));
    }
    else {
        return((*env)->NewStringUTF(env,"Not Installed"));
    }
}

if(!strcmp(appname,"Hummingbird Exceed")){
    if( RegOpenKeyEx( HKEY_LOCAL_MACHINE,
        "SOFTWARE\\Hummingbird\\eXceed\\CurrentVersion",
        0,KEY_READ,&key) == ERROR_SUCCESS) {
        if(RegQueryValueEx(key,"Version",NULL,&valuetype,ver,&versize)
            == ERROR_SUCCESS){
            printf("Found version %s\n",ver);
            RegCloseKey(key);
            return((*env)->NewStringUTF(env,ver));
        }
        return((*env)->NewStringUTF(env,"Installed but version not found"));
    }
    else {
        return((*env)->NewStringUTF(env,"Not Installed"));
    }
}
else {
    return((*env)->NewStringUTF(env,"Not Supported by Component"));
}
}

```

Appendix B: Update Scripts

Appendix B.1: Script to update an application

```

intf = "IDomainAdmin"
a = createComponent(intf)
locations = a.getMembers("Department 931B")
intf = "IUpdate"
n = length(locations)

```

```

dim status[n]
for (i=0; i < n; i=i + 1)
    b = createComponent(intf) at locations[i]
    status[i] = b.doUpdate("IBM Antivirus")
endfor

intf = "IDisplay"
home = "mymachine.watson.ibm.com"
c = createComponent(intf) at home
c.showList(status)
exit

```

Appendix B.2: Script to update an application with customizability

```

intf = "IDomainAdmin"
a = createComponent(intf)
locations = a.getMembers("Department 931B")
intf = "IUpdate"
n = length(locations)
dim status[n]
for (i=0; i < n; i=i + 1)
    intf2 = "IUpdatePolicy"
    d = createComponent(intf2)
    allow = d.isUpdateAllowed(locations[i])
    if (allow == false)
        status[i] = "Update Refused"
        continue
    endif
    b = createComponent(intf) at locations[i]
    status[i] = b.doUpdate("IBM Antivirus")
endfor

intf = "IDisplay"
home = "mymachine.watson.ibm.com"
c = createComponent(intf) at home
c.showList(status)
exit

```

Appendix C: Script to monitor a printer

```

printername = "colorful"
intf = "IPrinterManager"
pm = createComponent(intf)
location = pm.getLocation(printername)
intf = "IPrinterAssist"
pa = createComponent(intf) at location
while(true)
    iserror = pa.isError()
    if (iserror)
        err = pa.getError()
        shouldcorrect = pa.shouldCorrect(err)
        if(shouldcorrect)
            pa.correct(err)
            pa.setError(false)
        else
            notifyloc = pm.getNotifyLocation()
            intf2 = "INotifierGUI"
            gui = createComponent(intf2) at notifyloc
            gui.showMessage(err,printername,location)
        endif
    endif
endwhile

```


Accountworks: Users Create Accounts on SQL, Notes, NT, and UNIX

Bob Arnold – Sybase, Inc.

ABSTRACT

Accountworks is a system which allows any employee at Sybase, Inc. to use a web form to create accounts for new employees. Every new hire gets a personal account in SQL, Notes, NT, and UNIX administrative domains. Accountworks also creates initial stub entries in our SQL personnel database. It allows the user to make a number of initial choices for their new employee, including access to popular applications and whether to use Notes or UNIX email. Typically all new accounts are available within four hours after the web form is submitted. The system operates 24 by 365 to support our worldwide infrastructure. When the accounts are created, it guarantees a consistent, unique login, UID (for UNIX), Firstname.Lastname record, and password across all domains. It went into full production in July 1997, and has been used to create 1900 new accounts since then. Because this paper is intended to help anyone tackling cross-domain account management problems, it describes the architecture of Accountworks, the process of building it, numerous design decisions, and future directions of the project.

An Apology, By Way Of Introduction

There are a number of itemized lists in this paper, which will, probably, make for dry reading. However, it is hoped that they will also provide a valuable reference. If, at the beginning of the Accountworks project, we had started with a comprehensive set of issues, it would have helped us enormously. As it was, we had to muddle through as we discovered more and more questions that demanded answers. Given the complexity of the problem we tackled, and the limited space to discuss its solution here, the decisions are at least as important as the technical methods of implementing them.

Hopefully, this paper will be helpful to anyone who tackles similar problems. Certainly other sites will have other needs, and would make other choices. However, it seems likely that many organizations could use similar techniques to solve cross-domain account management problems. The descriptions of the Accountworks feature set, and the reasoning behind all these decisions, should at least serve to illuminate the many questions involved.

In The Beginning, There Was Mud

By early 1997, the process of bringing a new person into the company and putting all their necessary working environment in place was widely seen as a major problem. The infrastructure to support this process had not kept pace with the rapid growth of the company. Although some parts of the process worked well, they didn't always work together. In addition to regular employees, the company brings in student interns, contractors and temps; employees of our distributors and other business partners need accounts too. Everything from getting a phone to setting up super-user privileges for a system administrator was

taking far too long, sometimes as long as a month. Sometimes the hiring manager didn't begin the process until after their new person was already at work, which caused the predictable frustrations, phone calls, interrupts, emergencies, and escalations.

The Information Technology (IT) department is responsible for supporting most of this process. We have 7000 accounts in each domain, 15000 hosts, 100 locations around the world, and a WAN with links ranging from 28.8 modems to fibre to VPN. Our call-track system receives 10000 calls per month, many of which are linked to account administration.

In January 1997, a meeting with 40 interested people was held to fix the problems with the new hire system. These stakeholders helped define the overall project goals, and the group rapidly dropped to fifteen participants and a core of ten people.

Project Charter

Our primary project goal was to improve the process of enabling a new employee to become productive as quickly as possible. We took a broad view of this. We knew we would eventually manage the entire account-related life cycle of an employee at the company – we had to look ahead to termination and re-hiring issues. The account creation process had to work for contractors, temps, student interns, distributors and other business partners, as well as full-fledged employees. The charter included looking at, and sometimes re-engineering, other business processes related to hiring.

For example, early on in the project, we briefly considered building a semi-manual account creation process. Hiring one or two entry level staff to do nothing but create accounts would definitely have been cheaper in the short run. Such a solution had obvious

disadvantages though, in accuracy, speed, consistency, and data integrity. Furthermore it would still leave the IT organization as a potential bottleneck in the hiring process.

For a number of issues, we simply put documentation on the Accountworks web site. While short of a true one stop shopping solution, at least anybody could go to our web site to begin the hiring process. There, they would find all the necessary instructions, web links, and the Accountworks application itself.

One major change was the role of the Human Resources department in the new hire process. Our HR procedures vary from country to country, and sometimes among business units in the same country. Many of our European and a few North American business units relied on their HR staff to handle or coordinate many aspects of the new hire process, including the initial data entry. Our European IT operations depended on a fully enabled HR record to begin the account creation process. Accountworks required a fundamental business process shift, to make the hiring manager responsible for beginning the new hire process.

The other major process change affected some of the various help desks and systems administration groups around the world. Before Accountworks, half of these organizations were involved in the account creation process, occasionally in some cases and routinely in others. These processes were sometimes clearly defined, and sometimes not. Now it is crystal clear – none of these organizations have to do new hire account creation any more. The burden of the work is squarely placed in the ideal location – the person who cares about it most. And the person who cares, typically the hiring manager, has every opportunity to see to it that the job is done right – all they have to do is enter the correct data on the Accountworks web form.

Political Hurdles

In many respects the project was fortunate. We started with a number of advantages:

- The project was initiated and backed by new top management.
- With very few exceptions, the entire project team reported into the same IT organization.
- Everyone in the company could see the importance of the project.
- We had plenty of motivation – the project was an opportunity to fix our own long-festering problems.
- The core team had the necessary planning, architecture, programming, documentation, and user interface design skills.
- Some related systems, like our HR personnel database and calltrack systems, were SQL-based.
- We didn't have to actually manage these

domains, or even coordinate them, we just had to create consistent, unique, new accounts in them.

- A number of other simultaneous IT projects simplified our work:
 - Consolidation of roughly 90 NT security domains worldwide into three NT security domains.
 - Conversion of several MS Exchange email systems to Notes.
 - A UNIX home server consolidation project in Emeryville.
 - The only separately administered NIS subdomain was moved under the central NIS management system.
 - Consolidation of all desktop and laptop purchases into the IT department budget, instead of separate hardware budgets for each department.

We had a few disadvantages too.

The above consolidation projects, and other unrelated work, competed for staffing resources with the Accountworks project. Most of the core members were stretched thin, some of them chronically.

Years of neglect of each administrative domain had left them in a predictable mess. Clean up efforts simplified the project's work, but competed for the attention of project members. (Some clean up efforts were deliberately put off because they weren't required for the success of the project.)

Scope creep was a constant danger. We kept surfacing related issues which also needed to be solved. For each of these issues, we had to decide whether to ignore it, provide instructions and/or links to relevant web sites, or tackle it. Here are a few examples; many more came up along the way:

- How much of the entire new hire process should we really address? What about setting up the new employee's phone? ID badge? Building access? Company credit card? Network drop? Computer? What HW/OS should their computer be? Do they need more than one? (Various strategies were used.)
- Does the locations table in our personnel database reflect reality? (No, but we have to make sure it does.)
- Do we have to guarantee that logins are never re-used? (No.) What about guaranteeing unique UIDs? (Oh no, we didn't think of that, but we definitely have to do it.) Can UIDs be re-used after someone leaves the company? (Yes.)
- How do we handle accounts when we acquire another company? (These have to be handled on a case by case basis, so document a general strategy and put it on our web site.)

Top management originally thought the project would be quick and easy. Significant effort was

required to establish a more realistic timeline and staff allocation.

One of our core members was in Europe, a nine-hour time difference from the rest of the team in Emeryville, California. Coordinating our efforts with him was difficult. Two others, including our most important technical person, were in Ottawa; the three hour difference was more manageable. Two of these key participants had to travel to Emeryville for the roll out.

For some of our business units, HR had been responsible for the initial data entry for a new employee. HR staff naturally had concerns about making managers responsible, due to the potential for unclear process ownership and poor data entry. Management was not keen to assume new data entry duties at these locations either. With a lot of work and the backing of top management, we were able to work through these issues. Also, one of our core members from HR traveled to a number of offices around the world to address local concerns before the project rolled out.

Design Overview

What Accountworks does do:

- Allows any employee in the company to begin the process of hiring a new person, including automatic account creation, through an easy web form.
- Creates initial stub entries in our SQL personnel database for our Human Resources department to review and finish processing when all the approvals are received.
- Creates accounts in SQL, Notes, NT, UNIX, and upon request, a number of popular applications. These accounts are typically available within four hours.
- Guarantees unique, consistent login, UID, Firstname.Lastname records and passwords at the time when accounts are created.
- Creates calltrack requests for phone and equipment installation.

What Accountworks does not do (yet):

- Provide a multi-domain password changing tool.
- Provide an authoritative, automatically enforced, guaranteed-to-be-correct-across-all-domains database of what a person's login, UID, and Firstname.Lastname record is supposed to be.
- Manage accounts after they are created.
- Handle account terminations.
- Handle re-hires (because each person is supposed to have only one set of records in our personnel system no matter how many times they have left and returned to the company).
- Handle generic accounts.
- Handle large batch jobs of new hires (this

happens when Sybase acquires a company, brings in student interns, or a group of temps).

- Handle account changes, such as moving from one home server to another or moving between Notes and UNIX email.
- Handle login groups (NT Global/Local Groups, UNIX netgroup entries).
- Handle permissions groups (Notes groups, NT Global/Local Groups, UNIX group entries).
- Handle mailing lists (Notes mail groups, UNIX group aliases; luckily, auto-generated location-based UNIX email lists were already being handled by another application).

We lumped the last three items into the "general group problem," and decided that managing groups was too hard to do within the project deadline.

We built an application with six major components:

1. A web server front end, accessible to any employee, with instructions and links to related sites, and the web form which allows them to create accounts for new hires and to check the status of their requests.
2. The Accountworks SQL database, with the necessary knowledge of our environment to make intelligent decisions based on user input.
3. A large set of client tools to create accounts in SQL, Notes, NT, and UNIX; create calltrack requests for phone and computer setup; automatically grant access to some applications; send email or open calltracks to request access to other applications; report status back into the Accountworks database; and open service calltracks if any of these clients fails.
4. The Extraction SQL database which receives login, UID, Firstname.Lastname and other data from 34 data sources, and merges all that data into one table in the Accountworks database.
5. A set of programs to extract and parse the data from the 34 sources for the Extraction database.
6. Three client applications to administer Accountworks tables and help debug problems. These are accessible only by certain support staff.

Account Creation: A 12-Step Program

When someone wants to bring a new employee into the company, they go to the Accountworks web site. The first screen they see contains information, instructions, and a link to the Accountworks application itself. When they click on the link, the Accountworks data form comes up. Figures 1a and 1b detail the subsequent actions.

The **login** is used when creating accounts in SQL, NT, and UNIX. It is also stored as a "short-name" field in Notes.

The **Firstname.Lastname** record is used to create the Notes access key, which consists of Firstname,

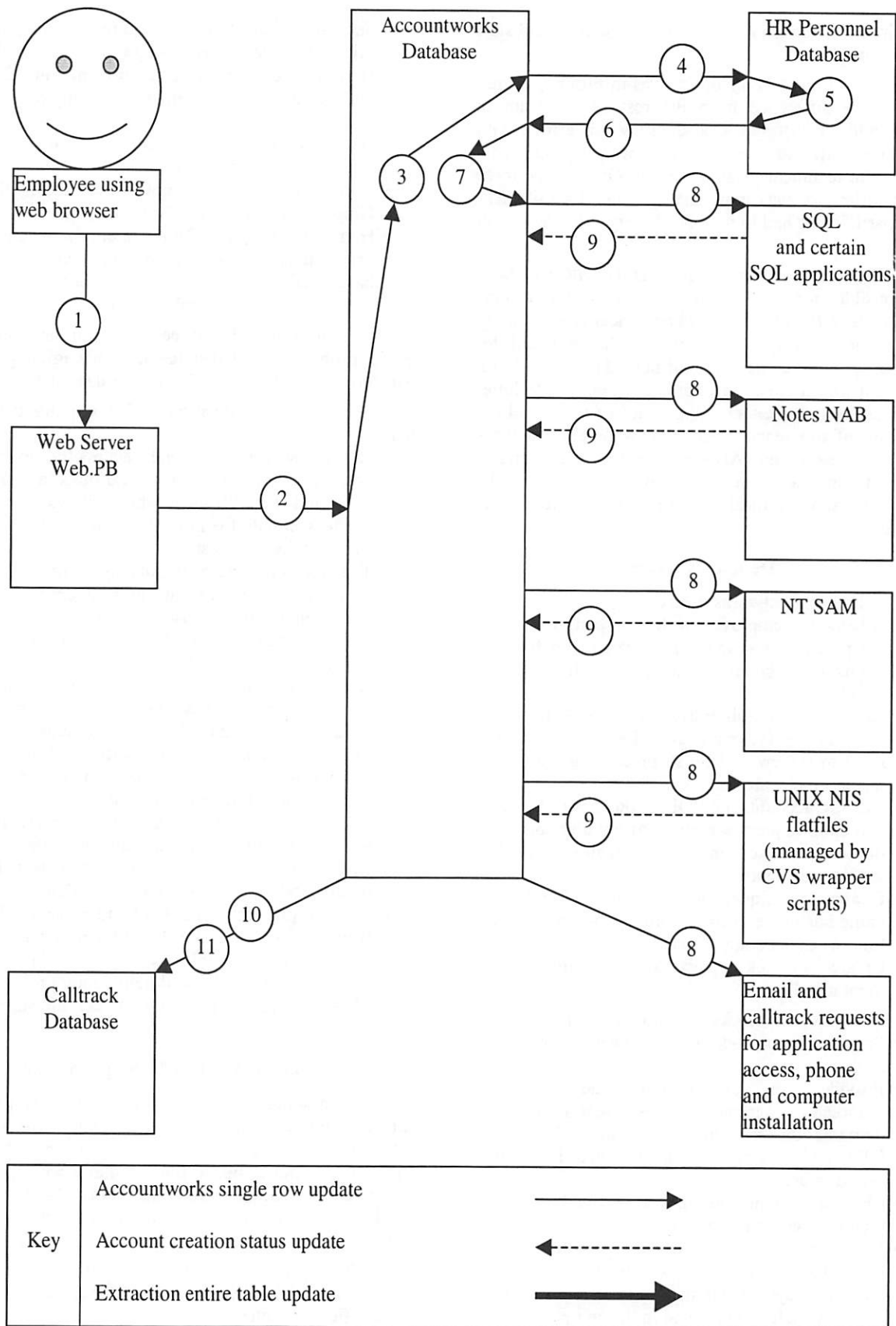


Figure 1a: Account Creation Process – Web interface, Accountworks database, and account creation tools.

1. The user enters the data for their new hire into the web form, and presses a Submit button. The data is written back to the web server.
2. The web server feeds that data to a stored procedure in the Accountworks database to begin the process of creating accounts.
3. The Accountworks database automatically generates a unique **login**, based on the new hire's name.
4. That **login**, together with other relevant information, is fed into a stored procedure in our personnel database.
5. The personnel database creates a new **emplid** (employee ID number) and a stub entry for the new hire.
6. The **emplid** is returned to the Accountworks database.
7. The Accountworks database tries to generate a unique **Firstname.Lastname** record. If it fails, it returns a web form to the user telling them what happened, and asking them for a different Preferredname (nickname) and a Middlename. When that is supplied, the process starts over, repeating until it succeeds.
8. Now the Accountworks database has all the information it needs to create an account in each domain. It uses the appropriate backend tools to do that for SQL, Notes, NT, and UNIX. It also opens requests in our SQL calltrack database for application access, phone setup and computer installation.
9. The account creation tools for each domain report back to the Accountworks database, describing their progress. Normally this all goes well, and each domain sets a "Complete_Success" status for itself.
10. If any of the backend tools report a "Complete_Fail" status, Accountworks opens a trouble ticket in our calltrack database for human intervention.
11. If 24 hours pass, and the Accountworks database sees that one of the domains has not reported "Complete_Success," it opens a trouble ticket for that domain.
12. The requestor, hiring manager and an optional third contact may check the status of the requested accounts at any time.

Figure 1b: Twelve steps to creating a signon.

Lastname, and an optional Middle_initial. The same data is stored in comment fields in the NT SAM and the UNIX passwd map. It is also used to create "login: Firstname.Lastname@notes-gateway" records in the UNIX aliases map for new hires who will be using Notes as their primary email system.

Troubles Come In Threes

There were three major problems that required solutions. Guaranteeing unique names for use by all systems was one. To solve this, we created the concept of an 'access key', which is an abstraction of the name which must be unique within a given system, and further must also be unique across all systems. Examples of 'access keys' are the UNIX logins from the NIS passwd map, email aliases from the aliases map, mailing list names in both SMTP and Notes, Notes ACL groups, NT username, and Notes login. We ended up with 34 different systems that needed to be synchronized by this concept of an 'access key.' A significant, beneficial side effect of this process was the identification of the systems and the ability to simply track (but not control) them from a single table.

Every evening, a set of scripts and stored procedures gathers access key data from the various sources, parses it into fields, and loads it into the appropriate tables in the Extraction database. Each data format, such as passwd and aliases file formats, has its own Extraction table. An hour later, we merge all the Extraction records into the "access_key" table in the Accountworks database. Each record in the "access_key" table knows its original data source and when it was first inserted.

When generating login and Firstname.Lastname guesses, Accountworks checks the "access_key" table to see if its guess is available. If so, that ends the guessing game. Otherwise, it moves on to the next guess. This is how we guarantee that any access key we generate is unique.

For example, our Extraction "passwd" table has four data sources. We gather /etc/passwd files from three important and representative UNIX hosts. These files only contain the typical system accounts like "root," "bin," etc. The fourth source is the flat file for our NIS passwd map, which contains 7000 records. It includes personal accounts for most of our employees, some generic accounts, but no "root" account.

Thus, the Extraction "passwd" table has three "root" records. All three "root" records are merged into the Accountworks "access_key" table. The "access_key" table also has a "root" record from the NIS aliases map (to forward mail from "root" to "postmaster"). A simple query against the "access_key" table will show us four data sources which use the "root" access key. If we ever hire a "Jennifer Root" or "Robert Oot," any one of these records is sufficient to keep us from creating a "root" login for them.

The second problem was collecting and modeling the data required to correctly map people to the correct login domains and home servers. It turned out that much of the information required to do this mapping, such as home server names/domain, office locations, and city to country mappings, existed in various databases, spreadsheets, and in many cases just a

person's head. Often the information was incomplete or inconsistent, and there was not a known master copy of the data. At one extreme, some offices have no home servers of any sort. At the other extreme, our Emeryville headquarters has perhaps 50 UNIX home servers, and numerous NT and Notes home servers too. Also, our personnel database has records for inactive locations as well as active ones, and we discovered that the locations data had not been well maintained. Once again, a significant side effect of automating the account creation process was the consolidation and cleanup of this required mapping data.

For each active location, we mapped three home servers: Notes, NT, and UNIX. A small office might have only a few PCs, or a few Suns. If a real local

home server could not be identified, we picked a home server in a more central office. The WAN topology dictated this choice, so we had to get accurate maps and information about this too. For example, our Dallas office has no UNIX boxes, so UNIX accounts for Dallas new hires were mapped to a Sun server in Chicago, the nearest WAN hub. Ditto for Notes. But the Dallas office does have an NT home server, which we used for the Dallas NT mapping.

Somewhat larger offices might have a few home servers, owned by various organizations. This forced us to create an organization pick list. For example, the technical support staff might be on one server, and everyone else might be on another. We created a "Tech Support" organization, and mapped new hires

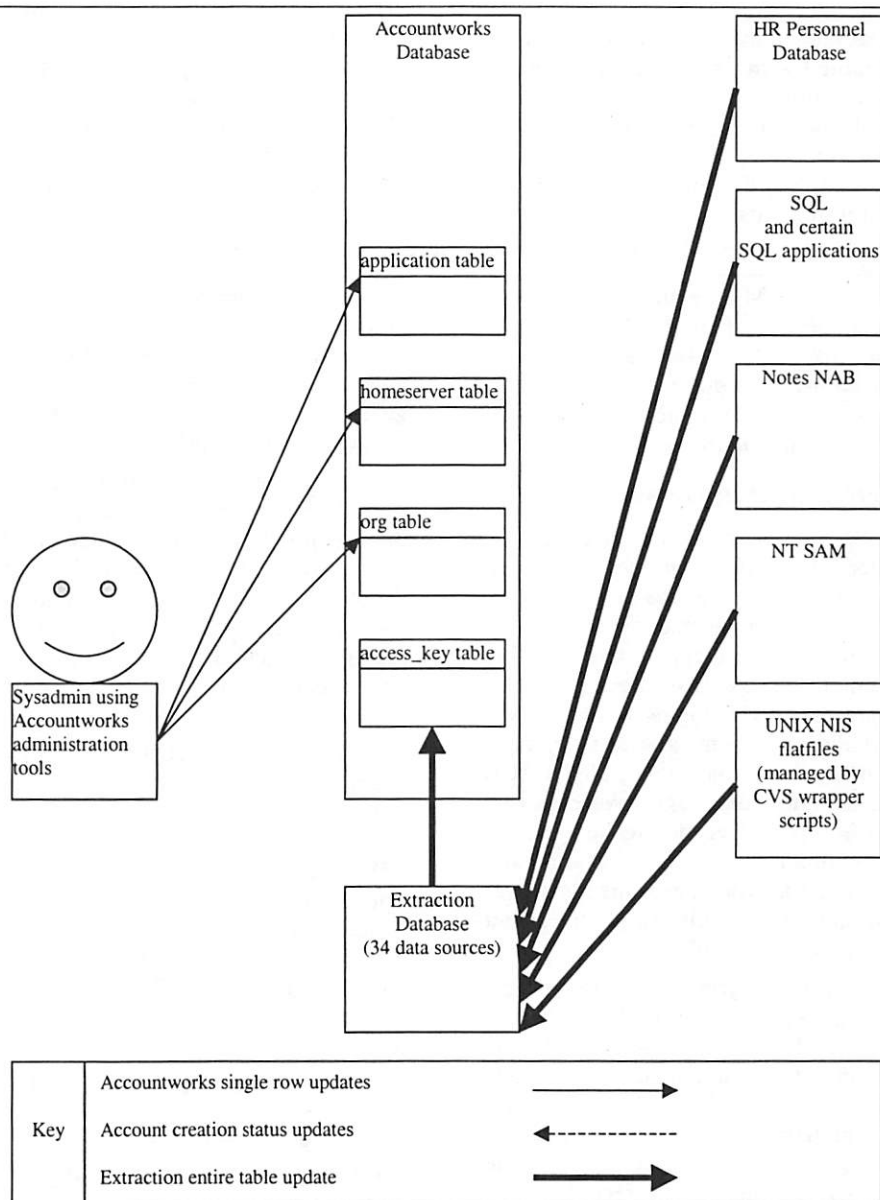


Figure 2: Extraction Database, Administrative tools.

for that location to their server; all other new hires would go on the other server.

Large offices might have many home servers, and even some departments are split between various servers. "IT Systems Administration" and "IT DBA" are on different UNIX home servers in our Emeryville headquarters. So we had to add a second level to the department tables. Appropriate rollups are done for sub-departments if someone chooses a department which doesn't have a specific home server at that location.

Although locations are hardwired to our personnel database, Accountworks "organizations," surprisingly, are not. Trying to track all the re-organizations and changes in department names and numbers had already doomed an earlier project to failure. Two of our core members had worked on that project, and kept us from making the same mistake. We decided that no matter what the official name of the department was, people could always identify with departments like "Sales" and "Engineering." This has proved a successful strategy.

It was a big help to know that the general direction had recently changed from splitting off new UNIX home servers to consolidating them. Even so, it took a surprising amount of time to come up with a mapping that would work. There were a number of reasons for this. One major factor was all the required research on the WAN topology and which locations were active or coming on line. Another was that it was hard to explain, or even remember, that we only needed to know where new hires would go now, not where everybody had been put in the past, and the new application would not move anybody's old home directory. In a few cases, we had non-UNIX machines providing NFS home services. But mostly, we had a delicate balancing act between adequately modeling the real world, and keeping the organization picklist small. We discovered the problem was complex enough that it was easier to interview key local sysadmins than request data via email. Our development centers, most of which have multiple buildings and a long history of creating a UNIX home server for every new department, were the hardest to model.

The third problem was the design of the request web form. Because someone might use it only once, we tried to make it as easy as possible to fill out. We minimized the amount of required information, and provided defaults, auto-populated fields, radio buttons, and pick lists wherever we could. We have only 16 input fields:

- First name*
- Preferred name (nickname)*
- Middle initial/name*
- Last name*
- Organization*+
- Location*+
- Notes/UNIX email*

- Start date
- Department number
- Job code (corresponds to a job title, not a specific opening)
- Company code+ (corresponds to country or business unit)
- Cube/Office
- Manager's login
- Alternate contact login (optional)
- Contact phone number
- Application requests+ (optional)

To make the web form easier to maintain, we drive pick list and checkbox creation with tables; these are tagged with plus signs (+) above. The fields marked with asterisks (*) are used for account creation; the others are necessary for personnel, contact, and equipment installation purposes.

What's In A Name?

A tremendous amount of time was spent on design issues surrounding names. Some of these decisions were easy, but others were not. Here are our choices, as they stand today:

- What names do we ask for, and how do we ask for them? We ask for four separate fields: Firstname, Preferredname (usually a nickname), Middlename, and Lastname. Our personnel system already required these fields, and it was impossible to reliably parse them out of a single "Name" field because so many firstnames and lastnames have embedded spaces in them (like "Mary Jo" or "van Beethoven").
- Should we ask for a middle name, not just a middle initial? Yes. When we initially put Accountworks into production, we only asked for a middle initial. Our Notes domain was the gating factor – it only allowed middle initials when creating Firstname.M.Lastname accounts, and no other system really needed a middle name. Since then, we have found it useful to accept a middle name for human eyes and paycheck records, so the middle initial field became a middle name field.
- Should the login and Firstname.Lastname record be the same across all systems? Yes, because that is simpler for everyone. (The answer could have been "No." Powersoft, before being acquired by Sybase, made sure that a person's PC login, UNIX login, and dialup login were all different. This was done for security reasons.)
- Should we use the SQL/NT/UNIX login as the access key for Notes, or the Firstname.Lastname record as the access key for SQL/NT/UNIX, so we could have a consistent access key across all domains? No, our Notes installation was too hard to change to use logins, and it was technically impossible to use

Firstname.Lastname records as SQL/NT/UNIX logins.

- Can we enforce correct capitalization of names? No. There are simply too many possibilities. For one person, "de Silva" might be correct, and for another it might be "De Silva." However, we do assume that two or more capital letters in a row are a typo, and convert them to an initial cap followed by lowercase. This mostly works, but not for Firstnames like "PT (Barnum)."
- How should we handle European or Asian character sets? Because the 7bit ASCII character set guaranteed portability across all four domains, we decided to convert 8bit alphabetic characters to 7bit, and not support double-byte characters.
- What special characters did we have to allow in the name fields? Hyphens, single quotes, periods, and spaces, for names like "Smith-Jones" and "O'Malley," "Joanie Caucus Jr." and "Peggy Sue."
- How do we make sure that people don't make typos, or use all capital letters? We can't. But we did add a confirmation screen to encourage users to make sure their data was correct, which helped a lot.
- How long could a login be? 8 characters. The UNIX login domain drove this decision.
- Could the login include the hyphens, periods, spaces, and singlequotes we accepted in the name fields? No. Hyphens had historically been generally discouraged in logins, and the others would lead to all sorts of technical trouble in many of the domains.
- Could the Firstname.Lastname record include the hyphens, periods, spaces, and singlequotes we accepted in the name fields? Periods, no; the others, yes. Firstnames like "E.T." would turn into "E.T.Phone-Home" which would run into trouble because of the double period.
- What are the valid characters for a login? Lower case letters and digits.
- How should we generate a Firstname.Lastname record? We actually try Preferredname.Lastname, then Preferredname.M.Lastname (if we got a Middlename). (We don't try the actual Firstname field, because many people use that only for legal and paycheck purposes.) If both of those are already taken by another employee or generic account, we ask the user to choose a different Preferredname or provide a middle initial. This is not a pretty solution, but it's as friendly as we can be, since the Firstname.Lastname record has to be unique. In such cases, we had to rely on the user consulting with their new employee.
- Should we let the user choose or request the login for their new hire? No. It was more work,

time was very tight, and we knew of cases where inappropriate login names had been chosen. We decided to automatically generate a unique login instead. This was a controversial decision, which continues to raise occasional questions.

- How should we generate the login? By using lowercase combinations of Preferredname, Firstname, Lastname, all the initials, and an appended digit if necessary.
- When we are generating the login and Firstname.Lastname record, should our checks for access key uniqueness be case-sensitive? No. If we somehow had used "jim.smith" already, a new "Jim.Smith" would have to choose another Preferredname or provide a middle initial.
- Do we have to guarantee that logins, UIDs, and Firstname.Lastname records will never be re-used after a person leaves the company? No, it's too late, it has already happened a lot. This was easier to implement, and is friendlier to the new hire because their preferred name is more likely to be available. The downside is that many in-house applications use login as a key (under the assumption that a login name would not be re-used), so occasional tweaks to these applications are sometimes required to keep the new hire from acquiring the attributes and history of an ex-employee.
- Could a new login name be the same as:
 - An existing login? No. Login names must be currently unique. (They do not have to be historically unique – see previous bullet.)
 - A mailing list name? No. Logins and mailing lists overlap in UNIX. If "all" is a mailing list, an "Allison L. Lucky" should never get "all" as a login.
 - A mailing list member? Yes. A member is either an external address (which is not a problem), or internal login name or mailing list name (which we already guarantee against conflicts.)
 - An NIS group? Yes. These namespaces do not overlap.
 - An NIS netgroup? Yes. These namespaces do not overlap.
 - An NIS hostname? Yes. A potential danger was that some local sysadmin groups used "login" as a hostname for DHCP clients. This procedure was changed to "login-pc" to avoid namespace overlap.
 - A Notes group? No. A Notes group can be used for permissions purposes and/or mailing lists, so the Notes group namespace can overlap with "Firstname Lastname" Notes ID's.

Security

We had to address a number of security issues, of course. Other security choices could have been made – there are tradeoffs for all of them. All of these security design decisions were implemented in the initial roll out; only two of them were changed based on our real world experience.

- Any employee can use Accountworks. More precisely, any person in our HR database can use it. We initially planned to restrict access to managers, on the theory that only managers would hire people. However, it turns out that in various parts of the company, technical/team/project leaders, supervisors, administrative assistants, and even contractors and out-source vendors bring in new employees. For some of our business units, HR had been responsible for beginning the hiring process for other departments. It soon became clear that the headaches of managing the authorization process would outweigh any potential security benefits. Besides, we were trying to enable the hiring process, not create another bottleneck.
- To use the Accountworks system, employees need a web browser that supports HTTPS, and they must enter their UNIX login and password.
- The requestor is allowed to request access to nearly twenty widely used applications for the new hire, via checkboxes on the web page. In some cases, these are granted automatically, and some are granted for new employees of the right department. Others require review, so call-tracks or email messages are generated to initiate these requests. The checklist does not include super-user privileges of any kind.
- A new employee's initial password is created by a random password generator. These random passwords set a good example for the new employee. However, they are also ugly, which encourages the new employee to change it (good), or write it down (bad).
- The same password is used when creating accounts on all systems. (After the new employee actually starts work, they can change their password on each system, of course.)
- The password, access key, and other account data are transported over the WAN in cleartext on well known ports, from the central Accountworks database to the machines controlling each administrative domain. This is in keeping with our general security model. However, the Accountworks machines are especially attractive targets. For that reason, login access to them is limited, file sharing access via SMB or NFS is limited or turned off, and they were among the first boxes to be attached by our Datacom group to switched ethernet hubs to make packet sniffing harder.

- Naturally, the new hire person needs to know their initial password. That means the password has to be stored on-line, so it can be retrieved. Very few support staff have direct access to the machine and database which stores the new hire passwords.
- Who should be responsible for retrieving the password and passing it to the new hire? Ideally, the person who cares the most about putting the new hire to work. So, the account requestor, hiring manager, and an optional third contact can log into the Accountworks status web page, check the status of the accounts, note their new employee's password, and pass it on to them.
- Every status page access is logged.
- If a prospective new employee eventually decides to turn down an offer from the company, our HR department initiates an employee "decline" procedure which removes all their Accountworks records and system accounts.

One major security dilemma centers around Notes. Access to a Notes database requires a Notes ID. This consists of a Firstname record, Lastname record, an optional Middle_initial record, a password, and a Notes ID file which contains the name records. Unfortunately, in the real world, people do forget their passwords. For many security systems, the standard fix is to have support staff reset the password, give the forgetful party their new password, and then tell them (or force them) to change it. Unfortunately, this has an ugly side effect in Notes. If the user has used Notes-encryption on any of their files, they can't decrypt those files any more, because resetting the password makes the Notes ID file out of sync with the database. Thus, Notes forces organizations to either a) abandon all encrypted files with forgetful owners, or b) store all Notes passwords so sysadmins can help forgetful owners retrieve their files. Long before Accountworks came along, our Notes administrators were storing the original Notes ID file, but not in the Notes default location. The project chose to continue that practice.

We are aware that complex systems are very hard to secure, and that a system's security is only as strong as the security of its weakest subsystem. Clearly, Accountworks is complex, and has many subsystems. The security implications are obvious.

Trouble In Paradise

There were, of course, a number of problems with the system when it was first launched, in spite of a lot of testing prior to going into production.

Testing is a tricky business. Using an isolated test environment is great for protecting the production systems, but sometimes it's hard or even impossible to recreate a realistic copy of a production system in a test domain. We used various hybrids of test and pro-

duction environments, which caused various problems.

For the UNIX domain, much of the testing was done against production systems. We got complaints from the user community about "Micky Mouse" and other silly passwd map entries created by the test data.

This wasn't a problem for the Notes domain, because we were unable to totally automate the creation of accounts by rollout, partly because of difficulties with the C language API for Notes. One person still had to press a few buttons to get the accounts created, and they exercised good judgement, so Notes users never saw the "Micky Mouse" accounts. On the other hand, the Notes process was slower than the others because human intervention was required.

For NT, most testing was done against an isolated test domain. But our production system has three NT security domains, and we realized shortly after we went live that we were only able to create accounts successfully in one of them. It took some time to get this fixed, so a number of NT sysadmins found themselves creating these accounts by hand. This didn't win the project team any brownie points.

Shortly after the rollout, it was realized that someone could create an account, get its password, use the first new account to create a second new account, and so on. Even worse, this method would allow someone who was leaving the company to create permanent dial-in access for themselves. So, we restricted Accountworks to accounts with fully activated HR records, and implemented a time hold before the password is released.

We initially designed the system to remove the password as soon as it had been viewed by the requestor, hiring manager, or the optional third contact. This caused more headaches than it was worth after rollout – too many users didn't actually remember the password they had seen, which meant phone calls to get the password reset, by hand, for each domain. We now have an automated routine which deletes the password after a period of time.

We had decided that the table of UNIX home servers would be validated against a comment field in our NIS hosts map, which had historically been maintained by our sysadmin staff. This turned out to be a bad idea, because responsibility for maintaining the validation data was too diffused. Each UNIX sysadmin is responsible for certain home servers, but because they didn't set one up very often, they sometimes didn't put the correct home server information in the NIS database. In such cases, the UNIX account creation script would refuse to create the account. We decided to turn off this validation, and focus the responsibility for maintaining the table of UNIX home servers on the Accountworks administrators.

We could have saved ourselves a lot of trouble if we had rolled out the initial version with a

confirmation screen. Our internal marketing efforts focused on the automated account creation benefits, not on the need for accurate data. Under the circumstances, some people entered test data just to see how well it worked. Other people entered real new hires, but they weren't particularly careful since it was just for system accounts, and they didn't know how much work it would be to fix the problems by hand. We added a confirmation screen to remind the user that they were creating real HR records, and to check their work before submitting the request. This helped a lot, but typos and incorrect data are still an occasional problem.

The various problems we had with the system at rollout had a domino effect. Some requestors would check the status, see that there were problems, and enter their new hire again. Even in the best case, we had to decide which records to delete from all systems. Other times, a sysadmin would fix a problem in one domain, without coordinating with other sysadmins or the Accountworks team. Also, the second request would often fail because the application could not create a unique Firstname.Lastname record. It doesn't matter how unusual or uncommon someone's name is – once their name is entered into the system, it's taken.

Support Complexities

Since Accountworks creates initial stub records in our HR database, this has relieved HR from some data entry work. But it has also created problems. HR staff has to delete records for prospective employees who never actually end up working at the company; this happens more often than it used to. HR staff has to correct bad data, such as typos in names, and delete records entered by people "just trying the system." This problem was particularly bad before we added the confirmation screen. Finally, HR staff have to delete test records entered by Accountworks application development and maintenance staff. Through all of this, our HR staff have been unusually patient and understanding.

Although the core technologies are SQL and web-based, many tools were used, particularly in the account creation and extraction scripts. Some of these are publicly available, including Perl [9], Sybperl [10], CVS [11], and the Systems Administration Environment [12]. Others are commercial: PowerBuilder, Web.PB, Transact-SQL, Adaptive Server Enterprise, Replication Server, and Open Server (Sybase, Inc.), FINAL (FastLane Technologies Inc.), Notes and NoteSpump (Lotus Development Corp.), Netscape Enterprise Server (Netscape Communications Corp.). A third group comes with other products: Bourne shell and friends (with UNIX), and isql (with Adaptive Server Enterprise). The diversity of the domains required a very diverse toolset.

The staff required to support Accountworks is small. Occasional operational problems can often be solved by junior support staff. Maintenance of organization, home server, and application tables requires minimal effort by trained staff. However, improvements and occasional problem debugging still require a diverse set of high skill levels. As of this writing, we have half a dozen more or less irritating bugs. None of them are critical, but most of them require a high skill level to fix.

Lessons Learned

When architecting the Accountworks application, our primary concern was data integrity. We knew all too well how messy our account domains were. If there was a way to foul up our namespaces, we had done it. We had been through numerous "final cleanups" before, but these heroic efforts were largely wasted without an automated system to keep the account domains in sync.

Therefore, we actively resisted statements like "We'll never hire a Robert Oot" or "That problem will never happen." Murphy's Law had struck far too often. The Accountworks database is highly normalized, with many integrity constraints. Wherever possible, we have tightly coupled our personnel database with Accountworks, using direct replication of tables. Entity relationships were rigorously defined with a conceptual modeling tool, which was then used to autogenerate the physical database structure. The web form is designed to minimize the possibility of bad data entry. Although we initially had a number of troubles around the edges of the application, the core database structure is clean and rock solid.

The SQL strategy has been a major win, because it enabled us to do this. In combination with several other projects, SQL is becoming the glue that ties our various management systems together.

Although Accountworks does not provide an automated system to keep accounts in sync, it is still a major step forward. New hires had been the major source of inconsistent account data. (The other three sources are rehire accounts, generic/system/test accounts, and human error.)

One concept which has been difficult to communicate to our user community, and even to our immediate coworkers, is that we still have no authoritative place to go to find out what someone's login or Firstname.Lastname record *should* be across all domains. Even the project architects didn't realize this problem until shortly before rollout, and we are a long way from having it completely fixed.

New account data is guaranteed to be consistent and unique only at the time it is created. The primary domains are still separately administered. Accountworks does not manage any of them. Thus, Accountworks is merely a multi-domain account creation tool, a glorious "adduser," if you will. Nothing prevents

authorized personnel from changing someone's SQL login, or the Firstname.Lastname record we keep in the NT SAM, or giving them a second personal UNIX account, or several entries in the aliases map. When someone changes their name, when they marry/divorce for example, every system has to be changed accordingly, by hand. One consideration is that access to old encrypted Notes documents is impossible for someone who gets a different Notes ID cut for them with their new Firstname.Lastname record.

The Extraction database is downstream from the personnel, SQL, Notes, NT, and UNIX account management systems (see Figure 2). Because of this, it can determine which domains are using which access keys, but it can't manage the account domains in any way. Except for the personnel data, it can't even tell, programmatically, which human beings (if any) are attached to which records. It can't tell if someone has an account or what its access key is. The lack of an automatically enforced authoritative account data system has proven to be a major headache.

Our ultimate goal is what we are now calling "Datamart." This project will define a set of authoritative data sources. To enforce that authority, we will automatically copy data from the authoritative sources to all downstream systems, including SQL, Notes, NT, and UNIX. When the Datamart project is complete, Accountworks will still be a front end to the various authoritative data sources.

Oh, Happy Day!

Everyone is quite happy with the progress to date, in spite of the initial rollout problems and remaining work. Our user community seems to have forgotten how far we have come – Accountworks is just part of the common toolset now. Sysadmins and help desk staff still have rehire, termination, and generic account issues to deal with, but these are much less disruptive and time consuming than our old new hire crises used to be. Naturally, many people can see ways to improve the system, but overall it functions smoothly and in many cases problems are fixed before the user even notices.

Finally, we have learned a lot. We have surfaced hidden problems, identified poorly designed systems, and examined dirty data sources. We are tackling them with various strategies. Although we still have a long way to go, we know where we are going and have a pretty good idea of how to get there.

Other Account Management Systems

Because of the need to integrate the administration of the four primary administrative domains (SQL, Notes, NT, and UNIX) with our personnel system, on a global basis, we were sure that no commercial product or public domain tools would meet our needs. An in-depth examination of one commercial product, and

technical meetings with other a few other vendors did not turn up anything we could use.

Account management solutions have been frequently published in the Large Installation and Systems Administration (LISA) conference proceedings. Eighteen papers were published on this topic in the first four years, and twenty-three total so far. Their requirements and methods, not surprisingly, were mirrored in many ways by our later work. A few quotes will illustrate what we have in common. The very first of these papers says:

"The solution at Athena was to create a central database of user information. The database is implemented in RTI Ingres and contains data on our users, courses and projects, clusters, the local systems, such as password files and mail aliases, are propagated from the master system several times a day. [...] For security reasons, the database resides on a restricted machine and can only be accessed directly by privileged users. Users and administrators access and modify the data through various utility programs." [1]

A centralized, secure, master SQL database, modelling our user community's needs, and accessible via external utilities – this summarizes some of our basic ideas nicely.

From the second LISA conference:

"We have (1) established a centralized Network Information Registry, (2) established ... policies ... and (3) designed a relational database to integrate the various administrative databases (including several Yellow Pages maps) and to reduce duplication of information. ... [W]hen a new account is created, the loginname and uid are checked for uniqueness in the NIR as well as in the YP passwd map and /etc/passwd file entries." [2]

The requirement for unique logins and uids, compared with multiple sources of this data, was critical to our own success. Again, we are following in other footsteps.

Two years later, the LISA proceedings contained this quote, which we could have taken almost word for word:

"The system selected had to meet several criteria, including:

- Centralized data storage
- Machine and vendor independence
- Flexibility in data to be stored
- Minimal changes to existing software
- Automated account installation
- Easy recovery from crashes
- Automated account deletion
- Simultaneous access for multiple users" [3]

Finally, the AGUS system [4], had we been aware of it, might have formed a foundation for some of our work. Here is the key quote:

"We wanted to use the same system to create accounts on UNIX, VMS, and Novell based

networks. The system should also be designed in such a way that it is simple to add additional system types to the configuration. For example, if the University decides to support user accounts on HP MPE systems, it should be relatively easy to extend AGUS to handle account creation under MPE." [4]

Here we have an extensible architecture which supports multiple non-UNIX operating systems. AGUS also embodies many of the design elements of earlier systems. For better or worse, it simply never crossed our minds that anything might already exist which came close to meeting our requirements, or which could be tailored to meet our needs with less work than building something from the ground up.

And, in the end, that is still true. The major differences between AGUS and Accountworks are:

- Trained support staff define account data prior to activation for AGUS; Accountworks builds account data on the fly.
- Both AGUS and our old system required a prior personnel record to create an account – this was one of the major bottlenecks that the Accountworks project had to fix.
- AGUS users request that pre-defined accounts in selected domains be built and activated; Accountworks users request that brand new accounts be created in all domains. Accountworks users also have to give enough information to make this possible.
- AGUS supports UNIX, VMS, and Netware on a few networks. Accountworks supports SQL, Notes, NT and UNIX; two email systems; and over 100 locations worldwide.
- AGUS is mostly written in C with a bit of Perl; the core of Accountworks is SQL based although many other tools were also employed.

For Accountworks, AGUS might have been able to help with the tools to build the UNIX accounts, although that was one of the easiest parts of the project. However, we still would have had to build the user interface; the database of logins, UNIX UIDs, and Firstname.Lastname records to guarantee uniqueness; and the intelligence necessary to configure accounts properly for each location and department.

Availability

Accountworks is not freely available. The company is interested in deriving value from this project. Please feel free to contact the author at rca@sybase.com for the current status of this effort or any related questions.

Roll Those Credits

Thanks to Paul Riddle, Paul Danckaert, Jack Seuss and Rob Banz for their email and conversations about AGUS. They provided useful information on the current status of the AGUS system.

Because of the complexity of the business processes and computer systems we were changing, many skill sets were required. Sixty or more people were involved in the implementation of Accountworks in one way or another. This core group was deeply involved with the design decisions and implementation:

- Jim Leask, Sybase Professional Services: Accountworks and SQL database architect, PowerBuilder Accountworks maintenance GUI tools, NT account creation and access key extraction scripts.
- Bob Arnold, Tools and Architecture Group: Accountworks architect, UNIX account creation and access key extraction scripts, NIS domain cleanup.
- Celeste Barker, IT Customer and Quality Services: Project Management, customer requirements.
- Jill Furman, Human Resources: Human Resources requirements and business processes.
- Bruce MacDonald, Tools and Architecture Group: NT requirements and planning.
- Eric Mittler, Team Notes: Notes account creation tools and access key extraction scripts.
- Chris Osterdock, Application Technical Services: DBA, SQL account creation tools, SQL and application access key extraction scripts.
- Geurt Schimmel, European Information Systems: Web-based Accountworks support tools, UNIX and NT account creation tools.
- Marcy Shaffer, Human Resources Operations: Web interface and calltrack programming.
- Sue Tran, Human Resources: Personnel operations and problem tracking.
- Shel Waggener, Response Center: Project sponsor, customer requirements.

References

The first four references have been cited in the paper. A few others of interest are also listed; references [7] and [8] are interesting because they see account management as part of a larger problem set.

- [1] Janet Abate. "User Account Administration at Project Athena." *Proceedings of the Large Installation System Administration Workshop*, 1987.
- [2] Deb Lilly. "Administration of network passwords and NFS file access," *Proceedings, Workshop on Large Installation System Administration*, 1988.
- [3] David Curry, Samuel D. Kimery, Kent C. De La Croix, and Jeffrey R. Schwab. "ACMAINT: An Account Creation and Maintenance System for Distributed UNIX Systems." *Conference Proceedings, Workshop on Large Installation System Administration*, 1990.
- [4] Paul Riddle, Paul Danckaert, Matt Metaferia. "AGUS: An Automatic Multi-platform Account Generation System." *Proceedings of the Ninth System Administration Conference (LISA IX)*, 1995.
- [5] Henry Spencer. "Shuse: Multi-Host Account Administration." *Proceedings of the Tenth System Administration Conference (LISA X)*, 1996.
- [6] Henry Spencer. "Shuse At Two: Multi-Host Account Administration." *Proceedings of the Eleventh System Administration Conference (LISA XI)*, 1997.
- [7] Dr. Magnus Harlander. "Central System Administration in a Heterogeneous Unix Environment: GeNUAdmin." *Proceedings of the Eighth System Administration Conference (LISA VIII)*, 1994.
- [8] M. A. Rosenstein, D. E. Geer, Jr., and P. J. Levine. "The Athena Service Management System." *USENIX Conference Proceedings*, Winter 1988.
- [9] Larry Wall, Tom Christiansen, Randall L. Schwartz. *Programming Perl, Second Edition*. O'Reilly & Associates, Inc., 1996.
- [10] Michael Peppler. "Michael Peppler's Home Page." <http://www.mbay.net/~mpeppler>. This web page contains links to Sybperl documentation, source, and related information.
- [11] Pascal Molli. "CVS BUBBLES." <http://www.loria.fr/~molli/cvs-index.html>. This web page contains links to CVS documentation, source, and related information.
- [12] Bob Arnold. "If You've Seen One UNIX, You've Seen Them All." *Conference Proceedings, Workshop on Large Installation System Administration*, 1991. See also <ftp://ftp.uu.net/usenet/comp.sources.unix/volume28/saenv-5.01>.

Single Sign-On and the System Administrator

Michael Fleming Grubb and Rob Carter – Duke University

ABSTRACT

Large organizations are increasingly shifting critical computing operations from traditional host-based application platforms to network-distributed, client-server platforms. The resulting proliferation of disparate systems poses problems for end-users, who must frequently track multiple electronic identities across different systems, as well as for system administrators, who must manage security and access for those systems. Single sign-on mechanisms have become increasingly important in solving these problems. System administrators who are not already being pressured to provide single sign-on solutions can expect to be in the near future. Duke University has recently embarked on an enterprise-wide single sign-on project. This paper discusses the various factors involved in the decision to deploy a single sign-on solution, reviews a variety of available approaches to the problem of electronic identity proliferation, and documents Duke's research and findings to date.

Introduction

The number of mission-critical computing platforms in today's enterprise is exploding. Applications that were once housed on "big iron" (MVS mainframes, enterprise-class Unix servers, etc.) are increasingly replaced with network-distributed client-server applications running across large numbers of smaller systems. Concurrently, the number of end users associated with these systems has increased dramatically in recent years. Together, these changes have resulted in a sizable increase in the number of disparate systems which end users interact with on a regular basis and which system administrators manage.

The forces driving these changes are well-known. Increased functionality and faster response time for end-users, increased demand for access to organizational information, decreased total cost of ownership, and Y2K considerations have all contributed to the explosion in network-based systems. While network-distributed applications have obvious advantages for both end-users and system administrators, their growth poses new problems for authentication, auditing, and security management.

In the traditional, host-based application environment, authentication was simplistic. Users were authenticated by the host operating system as they presented for entry into the system, and were then granted access to applications and data based on their locally-authenticated identities. Multiple applications co-located on a single large host system would share authentication information through the host's operating system, and access to the host system could be managed cheaply and centrally.

Audit trails could also be maintained centrally, and because all applications shared a common, operating system-specific authentication mechanism, could

easily be correlated between applications. Each individual would appear with a single identity across all applications on the central host. Therefore, audit records pertaining to an individual's actions within one application housed on the central host could easily be correlated with records pertaining to other actions taken by that same individual. Similarly, security and access management was straightforward, amounting in most cases to little more than maintaining a single user database and various application-specific authorization tables.

With the advent of network-distributed applications and the accompanying increase in the number of disparate enterprise systems, authentication and related issues have become more complex. Each different application or service may require separate authentication for its users. Users of multiple applications or systems may be confronted with a maddening list of disparate electronic identities and authenticators to remember. End users may be required to authenticate themselves multiple times during a single work session, and may have to remember and maintain a daunting number of login ids and passwords.

From the standpoint of the system administrator, the situation is no less disturbing. In the more distributed world, administrators must maintain authentication information across multiple platforms by creating, issuing, and deleting users' authentication information in multiple different contexts. While automated account management systems and distributed management tools (e.g., Tivoli, CA/Unicenter) can reduce the per-user effort involved in managing user identities across multiple systems, they are often as difficult to manage and maintain as the systems they support.

Further, administrators who manage electronic security for their organizations must often go to great

lengths to ensure that audit trails from disparate systems, regardless of whether they share authentication information with one another, can be reconciled. Actions performed by one individual across a number of different systems may be difficult to correlate with that single individual if the systems involved do not agree as to the individual's authenticated identity.

Additionally, the proliferation of electronic identities has social ramifications that can be devastating to overall system security. Presented with ten or more different authenticators for separate applications and systems, many end-users will resort to using insecure but easily remembered passwords, or will resort to recording their authentication information in an insecure fashion. At Duke, the authors have frequently run across this phenomenon in the form of users (and in some cases, system administrators) taping lists of their various logins and passwords to monitors in their offices. Enforcing any reasonable security policy in such an environment can be well-nigh impossible.

As end-users become increasingly frustrated by these issues, and as organizational leaders grow in their awareness of the serious security ramifications of electronic identity proliferation, pressure increases on systems administrators to provide technical solutions to these problems. Typically, this pressure results in the demand for a "single sign-on solution."

Single sign-on is something of a holy grail for large organizations. Everyone seems to want it, many people claim to have it for sale, but no one seems to agree as to what, exactly, it is. To many, "single sign-on" means that each user in the enterprise has only one userid and associated password. To others, "single sign-on" means that wherever a user logs in, he is presented with an interface and application set that is specifically tailored to him and which follows him throughout the enterprise. Another interpretation, favored by the authors, is that "single sign-on" *per se* is the goal of presenting the end user with only one authentication challenge during a single work session.

At Duke, the demand for a "single sign-on solution" started in earnest as the University began work on a number of parallel enterprise-wide computing projects. Payroll, human resources, student information systems, and purchasing were all targeted for migration off the institutional MVS mainframe and onto a set of distributed Unix-based systems. As the software engineers working on the deployment of these new systems began to consider work-flow patterns and information sharing requirements, and as users began to realistically consider the effect these new systems would have on their day to day activities, they realized the potential pitfalls in managing user information across such widely-distributed systems.

The authors, as lead system administrators for the arm of the university charged with the design and maintenance of institutional computing infrastructure, were approached with a deceptively simple question:

How should the University go about providing a "single sign-on" for the many and varied enterprise-wide applications in use and soon to be deployed on campus? A review committee, chaired by the authors, was formed and charged with answering this question. What was originally expected to be a three-month review process followed by rapid deployment of a complete solution has since become a 9-month-long investigation culminating in a recommended institutional strategy for user authentication which, we hope, will limit but not eliminate the proliferation of electronic authenticators across the enterprise. With this paper, we hope to offer other systems administrators the benefit of experience thus far at Duke in designing and implementing a comprehensive single sign-on solution.

Single Sign-On: One Password, Two Flavors

Before designing a single sign-on solution, a system administrator must first determine precisely what "single sign-on" means in her local environment. Is the demand for a single sign-on solution being driven by a need for enhanced security and auditability, or is it more an outgrowth of end-user frustration with the proliferation of electronic identities? To what extent are existing systems, many of which may already house multiple authenticators for individual users, to be participants in a single sign-on solution? Is the user population highly mobile, or do individual users work from fixed locations, and to what extent do individual users need simultaneous access to multiple different applications or systems? Answers to these and other basic questions will help focus efforts on solutions appropriate to local environments.

In particular, it is critical that organizations investigating single sign-on solutions decide whether their need is for a method by which to reduce the number of authenticators each end-user must remember and maintain, or whether their need is for a method by which to reduce the number of authentication operations a given user must perform in the course of a single work session. In the former case, the ultimate goal is the creation of a Single Authentication Realm, or SAR, where each user has only one authenticator (userid and password). In the latter case, the ultimate goal is the creation of an actual Single Sign-On, or SSO, mechanism, where each user authenticates only once during each work session. In the SAR case, individual users may be required to authenticate multiple times during a given work session, typically once for each disparate application or system accessed. In the SSO case, each user performs at most one authentication operation during each work session, although multiple authentication operations may be performed on behalf of the user by SSO software during the course of the user's work.

Building an enterprise-wide SAR is not a necessary prerequisite to an SSO solution, but in many cases it may be sufficient to meet an organization's

needs. A SAR can eliminate many of the security risks associated with electronic identity proliferation. With only one set of authentication information to remember and maintain, users are less likely to practice poor password hygiene and are more likely to protect their authenticator information. A SAR can also eliminate many of the system administration problems posed by userid proliferation. As each user has only one authenticated identity, reconciling audit trails across systems participating in the SAR becomes as easy as reconciling audit trails across multiple host-based applications on a single host, and user identity management (creating and removing authentication information for individual users) is reduced to managing data associated with the SAR. A SAR can also significantly reduce the overall cost of managing access to critical systems by reducing the number of times individual users must be initially certified (that is, the number of times individuals' actual identities must be ascertained and recorded in order to validate new authentication identities issued to them). With a functioning SAR, each individual need only be identified by system staff once (to authenticate their identity within the SAR at its inception), so strong certification policies can be enforced with less time investment by both end-users and system administrators.

A SAR, however, may not be sufficient to satisfy the demands of an organization's user population. While each user may have only one authenticator (e.g., a login/password pair) to remember and maintain, users may still be annoyed by the need to repeatedly enter authentication information during a single work session. At Duke, this has been of particular concern to the health-care community associated with the University's Medical Center. Physicians, nurses, and other health-care providers are understandably concerned about the time they may be required to spend repeatedly identifying themselves to different systems during a single session. In life or death situations, the few seconds a physician spends re-authenticating during a critical work session may make a real difference in the outcome of his or her patient's treatment.

Depending on exactly how it is implemented, a SAR may also increase the exposure of secure authentication information on possibly insecure networks, resulting in an overall *decrease* in system security. In essence, although each user need only remember one combination to open any organizational safe, the likelihood of that combination being compromised by a determined safe-cracker bent on watching the user's day-to-day activities may be increased under some circumstances.

Addressing these latter issues usually involves the deployment of a full SSO solution. This may take various forms: a carefully-designed SAR issuing reusable authentication credentials honored by all participating systems and services, a redesigned set of endpoint systems and applications built specifically to rely on a third-party authentication system to identify

users, a separate SSO application designed to proxy authentication operations for previously-authenticated users, or some mixture of all three. In addition to the advantages offered by its underlying (or implicit) SAR, an SSO solution can increase overall user satisfaction with computing systems, and can reduce the frequency and extent of a system's security exposure.

Depending on the specific SSO approach chosen, however, deploying and maintaining an SSO solution can be both labor-intensive for the system administrator and expensive. Maintaining the SAR and/or SSO infrastructure itself may in some circumstances become a significant drain on system administrators. Developing and maintaining interfaces between a given SSO mechanism and legacy applications or application environments may also be difficult or impossible. Additionally, cost considerations may limit the extent to which an SSO mechanism can be deployed enterprise-wide.

Application-Independent Authentication vs. Application-Dependent Authorization

It is important to note that both SAR and SSO solutions pertain to *authentication*, rather than *authorization*. Authentication, for purposes of this discussion, is the process by which the unique identity of an individual user is determined and represented in electronic form. Authorization, on the other hand, is the process by which a particular authenticated individual is identified as authorized to access a particular service or datum. As the authors found in early discussions at Duke with interested parties, authentication and authorization are frequently confused with one another, and as the authors also found in early discussions at Duke, authorization issues are frequently much more political than authentication issues.

Authentication is by its nature an application-independent process because an individual's identity is unique to the individual, regardless of what role the individual is fulfilling or what operation the individual is attempting to perform. Authorization is by contrast an application-dependent process because different applications may need to control access within their individual domains quite differently. User authorization poses its own unique set of challenges for both application programmers and system administrators, none of which are directly addressed by SAR or SSO solutions.

Authentication and authorization are, however, closely interrelated. In order to make appropriate authorization decisions, applications must among other things have access to reliable authentication information. An application which cannot uniquely identify its user cannot hope to make appropriate authorization decisions. Ideally, a SAR can provide the basis for building a cross-platform authorization infrastructure within an organization. While authorization mechanisms are outside the scope of this paper, it

should be noted that strong authentication mechanisms can play a part in supporting strong authorization mechanisms.

Options for Establishing a SAR

Once a decision has been made to implement a Single Authentication Realm, the system administrator may be tasked with recommending a particular authentication solution for use within the SAR. There are a variety of authentication options, each with its own strengths and weaknesses. Depending on organization-specific needs, any or all of the available options may need to be investigated or used in the development of an enterprise-wide SAR.

Policy-Based SAR: Solution by Fiat

Perhaps the simplest, although certainly neither the most robust nor the most functional solution the authors have heard of for building a SAR for a large enterprise is to dictate by organizational policy that each user shall use one userid and password for each system and application. Effectively, the organization achieves a SAR by fiat; institutional policy forbids the proliferation of authenticators.

In the authors' experience, such restrictive dicta are virtually impossible to enforce in any but the smallest and most tightly-managed organizations. As the number of users increases, and particularly as there are more system administrators who must cooperate with such a policy in order to make it enforceable, the labor required to enforce the policy increases beyond manageable limits.

Further, it is impossible to enforce such a restrictive policy across disparate systems without sacrificing password security; either those administrators responsible for enforcement of the policy must have access to users' individual authenticator information, or systems must somehow compare authenticators on a regular basis to identify violations of the policy.

Moreover, successful implementation of such a restrictive policy could increase the overall security vulnerability of systems managed under the policy by increasing the number of points from which the security of users' authenticators are vulnerable to attack. Because all cooperating systems must replicate users' authentication information locally, the security of all cooperating systems is limited by the security of the most loosely-managed system in the group. Also, since there is no single location from which all possibly-compromised authenticators owned by a particular individual can be modified or revoked, it becomes time consuming and expensive for system administrators to address emergent security breaches, and users run the risk of neglecting to update one or more compromised authenticators, leaving themselves open to continued attack.

Despite its weaknesses, this approach *can* be made to work under certain circumstances. A similar

approach has been in use within the Duke University Medical Center for a number of years. Upon arrival, each employee of the Medical Center is assigned a DEMPO (Duke Electronic Mail Post Office) id which is guaranteed unique within the set of DEMPO ids, along with a DEMPO password. System administrators throughout the Medical Center are required, by policy, to use DEMPO ids in creating accounts on their systems.

When the policy was instituted, in the late 1980s, the Medical Center had only a few multi-user systems, and still fewer system administrators. The majority of computing within the Medical Center was still centralized on a large MVS mainframe, and the majority of departmental computing platforms within the Medical Center were managed by a handful of corporate system administrators. Along with the policy, the Medical Center established an electronic mail service based on the PMDF product from Innosoft, and used a PMDF mail gateway to provide the appearance of a unified electronic mail system using DEMPO ids as electronic mail aliases. This unified electronic mail service made the DEMPO id mandate somewhat more palatable to end-users than it might otherwise have been, and given the limited number of administrators involved, the project was an initial success. Once the DEMPO id policy had become a part of the Medical Center corporate culture, its success as an organizational policy was a *fait accompli*.

A variety of complications have arisen over the years as a result of this policy. For example, DEMPO ids are assigned to end-users based on their surnames at the time of their hiring. Since many individuals change their surnames as a result of changes in marital status, adoption, etc., there are frequent requests to change users' DEMPO ids to match their new surnames, requests which cannot be honored under existing policy. Further, enforcement of the DEMPO id policy has been impossible in an environment in which individual principal investigators have almost complete control over their grant-funded operations, and each failure in enforcement leads to increased friction among those system administrators who are forced to follow the stated policy. Although the policy has been in effect within the Medical Center for a number of years, it continues to be the cause of repeated complaints by end-users and system administrators alike.

Unfortunately, this policy-based approach may often be the first one put forward by managers and administrators eager to arrive at a "quick fix" for the problem of identity proliferation. This method may, at first, seem like a low-cost, high-yield solution to what could otherwise be an expensive and complex problem to solve. At Duke, this was the case; it was originally suggested that enforcing a strict policy across systems might provide a simple way to control identity proliferation at the institution. We strongly discourage this

approach, which was ultimately discarded at Duke in favor of other, more feature-rich options.

Network-Based Authentication Mechanisms: A Better Approach

A more sound approach to the creation of a SAR is the use of a network-based authentication service of some kind. A number of different services are available to the system administrator, ranging in complexity from simple network-distributed databases to complex third-party authentication protocols based on various encryption technologies. Each of these approaches has its own strengths and weaknesses, and any or all of them may reasonably be viewed as candidates for partial or complete SAR solutions, depending on the specific requirements of an organization.

The simplest and most widely-utilized network-based solutions are network-distributed databases. By providing network-distributed access to a single authentication database, multiple cooperating systems can share users' authentication information, yielding a SAR. Some of the most common implementations of this approach are little more than network extensions of the well-known Unix *passwd*(4) table mechanism.

Hesiod: Authentication via DNS

One such approach, Hesiod, is a mechanism for distributing *passwd* table information (or virtually any arbitrary textual information) across a network using extensions to the traditional domain name service (DNS). Developed in the late 1980s at MIT, Hesiod builds on what was and still is a well-supported standard for network-based information distribution. In the Hesiod approach, authentication information traditionally stored in a local Unix *passwd* table (userids, encrypted passwords, etc.) is stored in extended DNS records and made available via extensions to the normal domain name resolution protocol (cf. RFCs 1034 & 1035). Hesiod-capable DNS servers support, in addition to the traditional DNS records of class "IN," records of class "HS." Class "HS" records may include records of type "TXT," containing arbitrary text strings indexed by other arbitrary text strings.

For a number of years, Digital Equipment Corporation's Ultrix operating system provided native support for Hesiod as a means of distributing Unix *passwd* table information. The DEC implementation of Hesiod was in use for some time within Duke's public Unix computing facilities, and presents a reasonable example of the Hesiod approach.

In this implementation, additional Hesiod pseudo-domains are constructed containing class "HS," type "TXT" records carrying user information in the standard Unix *passwd* and group table formats.

The primary DNS server for a given domain, "team.foo.org" for example, publishes not only the authoritative "team.foo.org" DNS information, but also authoritative Hesiod information for the pseudo-domains "passwd.team.foo.org" and "group.team.foo.org." These pseudo-domain tables contain standard *passwd* and group table information. For example, the "passwd.team.foo.org" domain table might include records of the form shown in Figure 1 that support access to user *passwd* table entries indexed on both login id and uid number.

This information, along with standard DNS information, is then made available across a network via an extended version of the standard name resolution protocol. An extension to the standard DNS API is made available through Hesiod-aware replacements for the standard resolver routines (commonly, a single "hes_resolve()" routine and a single "hes_error()" routine) to allow applications to search for authentication information in the form of *passwd* table entries. Hesiod queries for records pertaining to "rgc.passwd.team.foo.org" or "103.passwd.team.foo.org" would, in the example above, return *passwd* table information for the user "His Nobs."

In the DEC implementation of Hesiod, this ability to index individual *passwd* table entries within a Hesiod server's primary Hesiod tables on multiple keys was used to provide native support within the Ultrix operating system for Hesiod as a primary authentication mechanism. Under Ultrix, Hesiod-aware versions of the standard *getpwnam*(), *getpwuid*(), and *getpwent*() library calls were made available which would, depending on the configuration of a given system, return information retrieved via the Hesiod resolver API. Thus, a homogeneous network of Ultrix machines could achieve the benefits of a SAR through direct use of Hesiod.

Hesiod exhibits a number of strengths. Having been developed as part of the MIT Athena project, source code to implementations of Hesiod is freely available, and having been implemented as an extension to a well-known and well-standardized protocol, Hesiod boasts a level of standards compliance few similar systems achieve. Hesiod tables are relatively easy to manage, being structured similarly to regular DNS tables, and can be distributed across multiple Hesiod servers via the same zone transfer mechanism used to synchronize DNS tables between primary and secondary domain name servers. As is the case with traditional DNS information, Hesiod information can be published by multiple cooperating servers, some of which may act as "caching" servers to enhance Hesiod look-up performance across wide-area networks.

```
rgc HS TXT "rgc:PASSWORDCRYPT:103:4:His Nobs:/home/rgc:/bin/bash"
103 HS TXT "rgc:PASSWORDCRYPT:103:4:His Nobs:/home/rgc:/bin/bash"
```

Figure 1: Domain table records.

Hesiod is not, however, a perfect solution to the SAR problem. To date, the authors are only aware of one major vendor providing native operating system support for Hesiod as a primary authentication mechanism (Digital Equipment Corporation). Although it is possible to use the open Hesiod API to modify any arbitrary operating system or application to support a Hesiod-based SAR, retrofitting Hesiod support into existing applications (particularly vended applications) can be difficult.

Additionally, Hesiod does not in itself provide any mechanism for securing information stored in Hesiod tables. Not only are Hesiod tables stored in plain text on Hesiod servers, but Hesiod information is also transported over the network in plain text. Further, Hesiod servers do not necessarily enforce any limits on what client machines can gain access to Hesiod information, although recent versions of the BIND name server can be used to address that problem. These properties make Hesiod unsuitable for the distribution of secure information, and depending on the security requirements of specific organizations, unsuitable as the basis for a SAR.

NIS (YP): Authentication Databases via RPCs

Another network-based authentication database approach to developing a SAR is the use of Sun's Yellow Pages (YP) or Network Information Nameservice. Developed by Sun Microsystems under the auspices of the company's ONC development project, NIS (formerly known as Yellow Pages) provides much the same functionality as Hesiod, but uses a completely different network infrastructure. Although NIS was originally developed as a Sun Microsystems initiative, the subsequent opening of Sun's ONC RPC interface has led to a number of different vendors offering support for the mechanism.

In the NIS environment, as in the Hesiod environment, groups of cooperating machines are referred to as domains. Although they frequently coincide with DNS domains, NIS domains are completely orthogonal to DNS domains. Machines in multiple different DNS domains can, in theory, be members of the same NIS domain, and vice-versa.

Like Hesiod, NIS provides a mechanism for distributing arbitrary database tables (termed "maps") from a single set of database servers to a number of clients across a network. Also as is the case with Hesiod, the database tables distributed by NIS can include authentication information, typically stored in the standard Unix passwd and group table format.

NIS, however, uses a completely different mechanism to actually distribute information across the network. Whereas Hesiod relies on the pre-existing DNS standard for network extensibility, NIS relies on Sun's ONC RPC mechanism. In the NIS scenario, client machines within a given NIS domain use any of a variety of NIS-specific RPC calls to perform search, retrieval, and update operations on NIS data tables

stored on NIS servers. NIS clients are directed to their respective domains' NIS servers via a process of client binding. NIS client machines are intrinsically aware of the NIS servers in their environment, and direct any required RPC calls to their respective servers.

When used as the basis for a SAR, NIS takes the place of the traditional Unix passwd and group table mechanism, much as Hesiod does in the earlier example. NIS client systems are typically equipped with NIS-aware versions of the standard `getpwnam()`, `getpwent()`, and `getpwuid()` routines, allowing the use of NIS for network-based access to authentication information in a manner that is transparent to applications written to use native Unix authentication mechanisms.

As is the case with Hesiod, there may be more than one NIS server configured for a given NIS domain. In such cases, one NIS server acts as the master, processing all updates and periodically conveying new copies of its master tables to the other servers, which act as slaves. Both NIS master servers and NIS slave servers can respond to RPC requests, permitting multiple servers to be used both for redundancy and for load balancing. NIS is "open," in the sense that Sun Microsystems has released information about the NIS API and its associated network protocols, and various free versions of Unix and Unix-like operating systems include source code to functional NIS implementations.

Like Hesiod, NIS suffers from some serious drawbacks when viewed as a complete SAR solution. While NIS is supported "out of the box" by a wider variety of operating systems and applications platforms than Hesiod, they suffer from similar security problems. NIS implementations used as the basis for SARs typically still expose secure authentication information on a possibly insecure network, and may in some cases promiscuously distribute passwd table information to systems outside a given NIS domain. Such exposure is widely considered to constitute a security breach, since publication of user authentication information (particularly encrypted passwords) can assist would-be intruders in the exercise of brute-force and cryptographic attacks against the authentication system.

NIS+: Security at a Cost

Partially to address concerns regarding the openness of the NIS and Hesiod security models, Sun Microsystems developed NIS+, a NIS (version 2) follow-on. NIS+ provides some of the same features as Hesiod and NIS, but does so through yet another completely different network distribution mechanism. Although similar to NIS in name and general functionality, NIS+ represents a major shift in Sun's approach to network-distributed passwd table information.

Like NIS, NIS+ relies on an RPC mechanism to achieve network distribution of arbitrary data, and like NIS, NIS+ is designed specifically to solve the problem of extending the traditional Unix passwd table

mechanism (and similar mechanisms) into a networked environment. NIS+, however, relies on a modified RPC mechanism called "Secure RPC." In the traditional NIS model, arbitrary clients can bind themselves to any available NIS server and execute RPCs to access data housed on the server, leaving open the possibility of unauthorized accesses to secure authentication data. In the NIS+ model, clients and servers share a secret encryption key, allowing NIS+ clients and servers to cross-authenticate before information in any NIS+ tables is accessed. This additionally allows Secure RPC-based applications like NIS+ to take advantage of session-level encryption of network conversations, limiting or eliminating the exposure of secure information in plain text across a possibly insecure network. As a replacement for traditional NIS implementations in supported environments, NIS+ offers some significant security advantages.

NIS+ suffers from some serious drawbacks as the basis for a SAR in a heterogeneous network environment, however. To our knowledge, Sun Microsystems has not "opened up" the NIS+ protocols sufficiently to enable competing implementations. Thus, stable NIS+ clients are only available for use with Sun platforms (although note that there has been some limited success with reverse engineering NIS+ behavior for FreeBSD, Linux and other Free Unix platforms). While NIS+ servers *can* be operated in "NIS compatibility mode" to support non-Sun clients, running servers in compatibility mode erodes any security advantages offered by NIS+, making the NIS+-incompatibility-mode server little more secure than a traditional NIS server.

Further, NIS+ has been demonstrated to suffer from serious performance degradation and stability problems in very large environments. The authors have particular experience with these problems, having been tasked with managing a NIS+-based passwd table distribution mechanism for a collection of approximately 200 Solaris machines for over two years. At Duke, NIS+ is in use as a mechanism for distributing passwd table information (but not, as we will discuss later, actual authentication information) for some 37,000 Solaris users, and we have observed a number of serious problems with this rather large NIS+ deployment.

While running performance of NIS+ in the presence of no underlying server or network problems has not been a significant issue, serious problems have arisen when one or more NIS+ servers have failed. Specifically, it has been our experience that multiple NIS+ servers cannot be expected to function reliably in failsafe modes. In theory, NIS+ shares the traditional NIS feature of supporting multiple servers within a given NIS+ domain, each of which can respond to service requests from clients in the event of a failure among the server group. In practice, our experience at Duke has been that NIS+ servers frequently exhibit failure modes in which, rather than

refusing to respond to RPC requests and triggering client fall-back to other servers in the NIS+ domain, they respond *incorrectly* to RPC requests, resulting in clients receiving incorrect or invalid information. Further, because NIS+ (unlike NIS) offers no mechanism for forcing a particular client machine to direct its Secure RPC calls to a particular NIS+ server, we have found it to be exceedingly difficult to localize NIS+ failures when they occur, and nearly impossible to resolve them in an acceptable timeframe.

NIS+ has also exhibited serious administrative performance problems at times when it has been necessary to perform significant updates to large NIS+ tables. Transferring Duke's 37,000-entry passwd table from the local NIS+ master server to its slaves across a 100-Mbit FDDI ring has been observed to take as long as 15 minutes. More significantly, regenerating NIS+ credential information (required by the Secure RPC framework underlying the NIS+ service) for all 37,000 system users has been seen to take in excess of 12 hours real time on an 85-MHz Sparc5 workstation, during which time no access is available to NIS+ objects within the directory tree rooted on the affected server. Solutions to these problems recommended by the vendor have essentially amounted to replacing NIS+ with another service. NIS+, we have been told, was designed to support tables containing up to 10,000 objects, and is known to suffer performance degradation when table sizes increase beyond that limit.

These issues have driven us at Duke to work toward eliminating as many dependencies on NIS+ as possible, and lead us to recommend against NIS+ as the basis for a SAR in any heterogeneous or large environment. NIS+ *can* be used effectively as a replacement for NIS or YP in small, homogeneous environments, but does not scale well to large applications and is not appropriate for use in heterogeneous computing environments.

RADIUS, etc.: SAR by Proxy

Hesiod, NIS, and NIS+ are all SAR-like services based on the concept of distributing a single database of authentication information (usually a Unix-style passwd table) to a variety of possibly heterogeneous systems. Authentication is still achieved, with these mechanisms, through client-based look-up of authentication information and direct comparison of authenticators presented by end-users against those recorded in central authentication databases. While this approach has some advantages, among them interoperability with a plethora of existing applications designed around the traditional Unix security model, other approaches to network-distributed authentication are available.

One such approach, authentication by proxy, is exemplified by the RADIUS (Remote Authentication Dial In User Service) authentication mechanism. In the RADIUS scenario, authentication is achieved by clients passing their users' authenticators to a central

RADIUS server, which performs table look-ups to determine their validity, and returns an "accept" or "reject" response to the clients depending on the results of the RADIUS look-up. Client systems need not have direct access to any secure authentication tables, since actual authentication operations are performed by proxy on the RADIUS server. Rather than distributing the authentication database to client machines, RADIUS and related mechanisms pass individual users' authenticator information to a central server for validation.

RADIUS has been implemented by a number of vendors of remote-access hardware (terminal servers, routers and other network devices) as a cost-effective mechanism for providing user authentication from within embedded applications.

Depending on the specifics of particular implementations, RADIUS can suffer from a variety of security flaws as a basis for an organizational SAR. In particular, although the network path between a RADIUS client and a RADIUS server can be and frequently is hardened, the connection between a RADIUS client and its user is often unencrypted. As such, widespread use of RADIUS has traditionally been confined to applications in which the connection between the authenticating agent and the RADIUS client can be expected to be invulnerable to passive monitoring attacks (e.g., dial-up terminal servers).

A mechanism similar to RADIUS has been in use at Duke University for a number of years in support of authenticating dial-up access to the institution's campus-wide network. Users make dial-up connections to terminal servers on campus which, in turn, prompt for their authentication information and pass it to a Unix machine implementing a RADIUS-style authentication mechanism based on a secure third-party authentication service (Kerberos v4). Dial-up access to the terminal servers is either granted or refused based on the success or failure, respectively, of the Unix machine's authentication operation. While this poses serious security issues in the Duke environment, traditionally it has been felt that the risk of authenticator compromise involved in passing authentication information in plain text over a presumably secure telephone network and across the hardened Ethernet network connecting the terminal servers with the authentication proxy is more acceptable than the risk of allowing unauthenticated dial-up access to the campus network.

Because they are widely supported in certain niche-applications (terminal servers, etc.), RADIUS and related authentication proxying mechanisms are likely to play a role in any large organization's SAR. We do not, however, recommend that they be viewed as the basis for building a SAR, but rather recommend that they be used as an adjunct to other, more secure and more widely-applicable SAR solutions.

OTP: The Contrapositive of SAR

Yet another set of network-distributed authentication mechanisms which should be identified, perhaps less as SAR-building platforms than as competing solutions to many of the security problems addressed by properly-chosen SAR solutions, is the group of so-called OTP or One Time Password systems. Ranging in their implementation from totally software-based approaches (like S/Key) to totally hardware-based solutions (so-called "smart cards"), OTP solutions address the problem of identity proliferation in a somewhat radical manner.

Rather than striving to reduce a plethora of system- and application-specific authenticators issued to each end-user to a single, highly-secure authenticator, OTP solutions strive to make authenticators secure by changing them each time they are used. In the typical OTP scheme, a given user will have as many passwords (although hopefully not as many login ids) as he or she has authenticated work sessions. Each time a user's password (or, more generally, a user's authentication information) is used, it is immediately invalidated, and new authenticators are issued for the user.

Provided that the mechanism by which new authenticators are issued to end-users is sufficiently unpredictable to third parties, such OTP schemes can eliminate the security risks involved in sending plaintext authentication information over an insecure network. By the time a nefarious observer becomes aware of the user's identity and authentication information, it is no longer of any value.

Insofar as most OTP systems rely on some central authority to manage and coordinate the issuance and revocation of single-use authenticators, they may realistically be viewed as providing a sort of SAR, and to the extent that OTP systems maintain consistency of user identities (login ids, for example) through time, they can form the basis of something very similar to a SAR.

Software-based OTP mechanisms, such as S/Key, typically work by pre-assigning a sequence of authenticators to each individual, a list of "upcoming" passwords, as it were. Each user must periodically query the S/Key service for a new list of future authenticators, and must refer to the list whenever performing authentication operations on supported systems. Similarly, all supported systems must be accessible at an administrative level to the S/Key service, so that authenticators may be invalidated and replaced as they are used.

Obviously, such list-based OTP solutions can only be as secure as the mechanisms by which these lists of authenticators are transported. Requesting a new authenticator list over an unsecured network channel, printing an authenticator list on an insecure printer, or storing an unencrypted list of future authenticators online or in some other insecure fashion undermines the security of the entire list. So long as

sufficiently secure channels are used to manipulate authenticator lists, however, list-based OTP mechanisms can provide an inexpensive enhancement to the overall electronic security of an organization.

By contrast, hardware-based OTP solutions (like Security Dynamics' SecurID product) typically generate new authenticators as they are required, thus eliminating the need for persistent lists of single-use authenticators. In these schemes, each user is issued an electronic device (typically a roughly credit-card-sized microcomputer) equipped with enough intelligence to calculate a complex and unit-specific cipher which is in turn used as the owner's authenticator. Depending on the specific implementation, the cipher may be a function of the current time (in which case the user's "smart card" must have a means for synchronizing its clock with the central security system), a function of some random challenge string presented to the user at the start of an authentication operation (in which case the user's "smart card" must offer some means for ascertaining the value of the challenge string), or a function of both. Additionally, smart card systems may require users to provide the interface between cards and authenticating systems (by manually responding to OTP challenges presented by target systems) or may make use of additional hardware to directly connect authenticating systems to smart cards.

Provided that the cipher calculated by the user's smart card is sufficiently secure (that is, provided that it is both cryptographically secure and sufficiently difficult to reproduce using other hardware) such systems can offer the same sorts of advantages list-based OTP solutions like S/Key offer. In addition, hardware-based systems offer the advantage of requiring little or no change in the behavior of end-users; users need not learn to operate a new software system in order to manage their authentication information. Rather, they may simply substitute a password provided on demand by their personal authentication device for a traditional remembered password.

Hardware-based OTP solutions typically involve much higher installation and deployment costs than software-based solutions. Especially in large organizations, the cost of acquiring and issuing thousands of smart cards and possibly card reading interfaces may be prohibitive, and in addition, the long-term cost of maintaining and replacing malfunctioning or stolen authentication devices can be exorbitant. Similarly, although software-based OTP solutions may be deployed without significant capital outlay, even within large organizations, the cost associated with retraining users and supporting their use of a more complicated authentication scheme may be prohibitive.

Unlike distributed database SAR solutions and proxy solutions such as RADIUS, OTP solutions rely on "something the user has," rather than "something the user knows" to achieve strong user authentication. Depending on the extent to which users protect their

OTP lists or smart cards, such "bearer bond" authentication strategies can be either more or less secure, overall, than equivalent password-based systems. Increasingly, OTP mechanisms are being viewed as an adjunct to, rather than a replacement for more traditional password systems. While the cost of deploying an OTP solution ubiquitously across a large organization may be prohibitive, the cost of deploying such a solution for a few, key users within the organization (presumably those whose level of authorization makes the strength of their authentication particularly important) may be less so. In many instances, hardware-based OTP solutions are being deployed *in addition to* traditional password-based mechanisms in order to achieve an even higher level of certainty in authentication than either mechanism can provide alone.

PKI: SAR Meets Star Wars

Yet another, arguably much more secure approach to network-distributed authentication involves the use of digital certificates under the auspices of a public key infrastructure, or PKI. Although it has yet to achieve the status of a true "standard," the most widely-accepted approach to digital certificate/PKI authentication is the proposed X.509 standard.

In this scenario, authentication is achieved by presenting a digital certificate to the challenging system or application. This digital certificate is actually a structured block of data identifying the certificate's owner and typically including the owner's public key which has been digitally signed (using one of a number of related public-key encryption technologies) by the certificate's issuer, the user's certificate authority or CA. Presented with this digital certificate, a cooperating system can verify (by decrypting the certificate with the CA's public key) that the certificate is genuine (i.e., that it was issued by the certifying authority). The CA can frequently be queried to verify that a particular certificate is valid (i.e., that it has not been revoked). Certificate authorities, in this scenario, must each have their own public key/private key pairs (to effect signing of the certificates they issue), and must act as key distribution agents, providing a central repository for the retrieval of public key information. Participating systems and applications then "trust" particular certificate authorities, accepting valid certificates issued by those CAs, usually on the basis of the CA's being known to use trusted methods to identify individuals before issuing them certificates.

Public key certificates offer a number of advantages as an authentication mechanism. Being based on public key encryption mechanisms, certificates are highly cryptographically secure. Forging a public key certificate without prior knowledge of both a user's private key and his CA's private key is virtually impossible. Further, certificates offer some of the advantages of hardware-based OTP systems, in that authentication is achieved based on a user's

possession of something (his or her certificate) rather than his knowledge of something (his or her password).

Additionally, digital certificates provide a natural means for engaging in secure communications over a possibly insecure network. Once a user's public key certificate has been exchanged and validated as proof of authentication, a shared encryption mechanism is available to both the authenticating user and the system to which he or she is authenticating. The user may encrypt data in his or her private key (as distributed by the user's CA) to ensure the authenticity of transmissions (data encrypted in the user's private key can only be decrypted using that user's public key). Participating systems may encrypt data in the user's public key, ensuring that their transmissions can only be decrypted using the user's private key.

Public key certificate systems are not, however, a perfect solution to the SAR problem. In order to deploy a SAR based on public key certificates, an organization must first develop and deploy a rather complicated set of support services, a public key management infrastructure, which may include, in addition to one or more local Certificate Authorities, a key escrow system (for the secure storage and retrieval of private key information) and mechanisms for updating, invalidating, and re-publishing public keys. Security of the various portions of the public key infrastructure becomes critical, since compromise or impersonation of a part of the public key infrastructure can directly undermine the security of any authentication mechanisms designed around it.

In certain environments, it may be feasible to rely on an external CA, effectively outsourcing the work involved in setting up and maintaining a public key infrastructure. In many organizations, however, the need for guaranteed access to the certificate authority or special requirements for user identity verification and certificate maintenance (frequent certificate revocation, for example) may make an off-site CA untenable. On the downside, reliance on an organization-specific CA may cause difficulties for users who interact with systems which do not trust the organizational CA, resulting in individuals having to juggle multiple personal certificates (one for use within the organization, for example, and one for use outside the organization) and undermining the intent of a SAR.

Moreover, few applications and no common operating systems are currently built to support PKI certificates as the basis for user authentication. So-called "personal certificates" are supported by a wide range of web-based applications via the intrinsic support for certificates provided through common web browsers (Netscape, Internet Explorer), but non-web-based applications and systems typically offer no mechanism for supporting certificates as an authentication tool. A SAR based entirely on the exchange of

public key certificates in any but the most homogeneous of organizational environments would require significant re-development of common applications and systems.

PKI certificates further pose problems in environments where users are highly mobile. Because of their complexity and size, certificates cannot be feasibly memorized or re-entered by their owners, and so must be stored electronically. In non-mobile user environments, certificates can reasonably be stored on users' preferred client machines, where they can be reliably accessed at all times. In more mobile environments, where a single user may work from any of a number of locations, access to a user's PKI certificate can become more complicated. Hardware-based solutions, in which certificate information is stored in "token cards" carried by users, can make certificates easily transportable, but are expensive to implement across large organizations. Software-based solutions involving key and certificate escrow systems can make certificates network-accessible, but require some form of alternative authentication in order to ensure secure access to escrowed key/certificate information. These difficulties are specific, of course, to end-user authentication systems; public key certificates used to authenticate systems to one another need not be transported from one system to another and so do not suffer from these complications.

In spite of these difficulties, it is clear that public key certificates will play an increasingly important role, particularly in the realm of secure electronic commerce applications. Any SAR or SSO infrastructure developed today will need the ability to use public key certificates where they are appropriate. However, building a SAR based solely on public key certificates and a central CA will only be feasible for the short term within certain organizational environments. The authors believe that the cost of deploying, managing and maintaining a public key infrastructure currently makes certificate-based authentication mechanisms unattractive from the point of view of the typical system administrator.

Kerberos: Established Technology, Secure Authentication

Yet another secure network-based authentication mechanism, and one which the authors recommend as the preferred basis for developing SARs within most organizations, is Kerberos. Developed at MIT under the same Athena project which spawned the Hesiod system discussed earlier, Kerberos has been in use within a large number of organizations for a number of years. The Kerberos technology, represented by MIT Kerberos versions 4 and 5 as well as by the security service associated with the Open Group's DCE infrastructure, is both widely-understood and reasonably impervious to common methods of attack.

The impetus for the development of the Kerberos authentication model was a simple question: how can

users be authenticated to systems across insecure networks without exposing their authentication information to would-be attackers monitoring network traffic? Relying in part on previous work by Needham and Schroeder, developers at MIT designed the Kerberos model as a means of authenticating clients without resort to passing re-usable authentication information (passwords, etc.) over an insecure network.

In the Kerberos approach, clients and servers are loosely grouped together into "realms," with each realm containing at least one shared security server. This security server, called a KDC or Key Distribution Center, shares a secret encryption key with each authenticatable user and with each participating network service provider within the realm. These keys are used as shared secrets to effect user authentication and to issue reusable authentication credentials called "tickets." Users and service providers within a given realm trust the realm's KDC, and in some cases, may trust foreign KDCs through a chain of trust relationships and shared keys between KDCs in multiple realms.

Kerberos achieves authentication through the use of shared secret keys. The model is based on the simple realization that if two parties wish to verify one another's identities, they may do so securely by exchanging information strongly encrypted in a shared secret key. The party initiating the authentication process can send a message encrypted in the shared secret key, and if the responding party is able to properly decrypt the message, both parties can be reassured of one another's identities. Provided that the key is actually a secret shared solely between the two parties, the ability to decrypt one another's messages is sufficient to prove authentication. If part of the encrypted information passed in the original exchange is further encrypted in a secret key known only to the originating party, the originating party can subsequently verify the origin of the initial message.

In reality, the Kerberos model is made more complicated by the need for more than two parties to authenticate to one another securely in real environments (necessitating the use of persistent "tickets" as records of prior authentication), and by the need to address certain security vulnerabilities intrinsic to its use in insecure network environments (network replay attacks, dictionary-based key-guessing attacks, etc.).

Conceptually, the KDC can be viewed as providing two different but related services: an initial "authentication service" or AS, used to perform initial authentication of users and issue tickets for the ticket granting service, and a ticket granting service or TGS, used to issue tickets for other participating services. In most existing implementations of the Kerberos model, the AS and TGS reside on the same secure host(s), and in most cases the two are implemented using the same executable code.

In the classical Kerberos authentication model, initial authentication involves the acquisition of a "ticket granting ticket" or TGT (basically, a ticket for the ticket granting service) from the KDC's AS. This is accomplished in two steps. In the first step, the client sends an authentication request to the AS, indicating the user for whom a ticket granting ticket is sought and providing information (time stamps, etc.) useful for validating the request. The AS responds with a TGT which is encrypted in the user's secret key. The client's ability to decrypt this TGT is the basis for the client's proof of authentication, since only the user and the KDC are presumed to know the user's secret key. Encryption is usually achieved using the DES encryption algorithm, but can in theory be achieved using any symmetric-key encryption scheme.

Enclosed in the ticket granting ticket are a number of pieces of information, of which five are particularly important: an actual user credential identifying the authenticated user (encrypted, itself, in the server's own secret key, to prevent foreign construction of credentials), a timestamp issued by the KDC identifying the time at which the ticket granting ticket was issued, a lifetime value indicating how long the ticket is to remain valid, a checksum useful for ensuring that the ticket has not been modified from its original content, and a randomly assigned "session key" for use as a shared secret between the now-authenticated user and the KDC. If the client is able to decrypt the ticket granting ticket correctly (i.e., if the resulting ticket contains a valid checksum) and if the timestamp on the ticket matches (within a short window) the current local time on the client, the ticket is accepted as valid. If any but the correct key is used to decrypt the ticket, or if the data within the ticket is changed from what was originally issued by the KDC, the resulting decrypted key will not match its checksum and the ticket can be identified as invalid. Together, encryption in the user's secret key and the presence of the checksum ensure the integrity and confidentiality of the TGT.

At no time during the initial ticketing transaction does the user's secret key information pass over the network, and at no time do reusable authentication credentials pass over the network unencrypted. As such, the initial ticket exchange can in principle be performed securely over an insecure network.

Once acquired, the user's ticket granting ticket can subsequently be used in additional ticket exchanges with the KDC, usually to acquire so-called "service tickets" as proof of authentication for use with services other than the TGS. Subsequent ticket exchanges proceed similarly to the initial ticket exchange, with requests being made to the TGS rather than the AS, and with requests specifying a target service in addition to a target user. Authentication for service ticket requests may be performed in the same way as authentication for initial ticket requests is performed (using the client's knowledge of the user's

secret key to authenticate the user), but more commonly, a previously-acquired TGT is used as proof of authentication. The TGS responds to a service ticket request with a service ticket containing information similar to that in the TGT. The response is encrypted in the session key shared between the KDC and the authenticated user, and contains within it information encrypted in the secret key shared between the target service and the KDC. This service ticket can then be used as part of an authentication transaction with the target service, which can ensure that the ticket being presented to it is valid based on its being encrypted in the target server's secret key (a key known only to the target server and the KDC).

Three different implementations of the Kerberos model are currently in common use: the "original" Kerberos version 4, Kerberos version 5, and the Open Group's DCE security server. All are built atop the general framework outlined above, although each differs from the others in the specifics of the communication protocol used and the details of the contents of tickets. Kerberos version 4 is perhaps the most commonly-deployed implementation of the model, having been generally available since the mid-1980s. As Kerberos V4 came to be deployed on larger scales, it became apparent that the original implementation suffered from some serious security flaws, including a particular susceptibility to dictionary-based attacks. Kerberos V5 was developed in the early 1990s to address these concerns, and to provide some additional features (e.g., better cross-realm authentication, forwardable tickets, and renewable tickets) required by new applications of the model. The addition of support for prior encryption of ticket granting requests and changes in the underlying ticket exchange protocol make Kerberos V5 less vulnerable to certain common dictionary-based cryptographic attacks. The DCE security service was developed as part of the Open Group's Distributed Computing Environment, a larger (and some have said, too large) project striving to provide an infrastructure for secure, cross-platform distributed computing. Loosely based on an intermediate version of Kerberos V5 from MIT, the DCE security service is conceptually similar to Kerberos V5, but relies on the DCE "secure RPC" mechanism as the underlying conduit for ticket exchanges.

Kerberos offers a number of positive features for both system administrators and end-users as the basis for an organizational SAR. The Kerberos model provides the advantages of strong authentication based on strong encryption without the overhead and complications exhibited by current PKI implementations. In principle, only one dedicated network server must be installed, secured, and maintained to support a Kerberos infrastructure, although in practice multiple "clone" security servers are usually deployed to provide redundancy and availability. Like PKI/certificate based authentication mechanisms, Kerberos provides a natural mechanism for ensuring the privacy and

integrity of application-level data exchanges over an insecure network (via the session keys distributed with Kerberos tickets), and provides a mechanism for bipartisan authentication (i.e., the model supports both the authentication of client users to server systems and the authentication of server systems to client users).

Kerberos boasts an enormous installed user base, and is one of the most proven network-based authentication schemes available. From the standpoint of the systems administrator, who may be held ultimately responsible for both the security and availability of systems participating in the SAR, this historical track record can be a significant advantage. Kerberos is further supported by well-established Internet standards, and as such forms the basis for continuing development efforts world-wide. New implementations of the Kerberos model continue to be developed (viz., work at KTH in Sweden and recent work toward both Tcl and Java-based implementations of Kerberos V5). Kerberos is already supported as an authentication mechanism by a number of high-profile application vendors (Oracle, SAP), a variety of common open application servers (IMAP, POP, ACAP) and is supported natively on a number of operating system platforms (AIX, Solaris). Upcoming versions of some very popular operating systems (Windows NT, Novell Netware) are also expected to include native support for Kerberos as an optional authentication method.

Kerberos does suffer from some well-known deficiencies, both as an authentication system and as the basis for an organizational SAR. Kerberos is, at its base, a password-based system, and as many have pointed out, it is subject to a variety of password-guessing attacks. Further, the model (particularly in its implementation under Kerberos V4, but to some extent still in Kerberos V5 and the DCE implementation) is subject to certain types of replay attack; a determined attacker may, under certain circumstances, be able to circumvent the replay protections built into the Kerberos protocol and for a short time masquerade as an authenticated user by replaying part of a previous ticket exchange on an insecure network. As Bellare and Merritt, among others, have pointed out, Kerberos is also subject to environmental attacks. Depending, as it does, on time synchronization between participating clients and servers, Kerberos security can be undermined in the presence of insecure timekeeping mechanisms. Additionally, since Kerberos session keys may be used to encrypt multiple messages between a single client and server during the lifetime of a given Kerberos ticket, session keys may be vulnerable to cryptographic attack by sufficiently determined enemies.

Like PKI solutions, the security of a Kerberos-based SAR is wholly dependent on the security (both logical and physical) of the systems making up the authentication infrastructure. The security of the KDC within a given realm is as important in the overall security of a Kerberos-based SAR as the security of

any portion (CA, key escrow systems) of a shared PK infrastructure. The authors believe that, because of its relative simplicity and wide availability, a Kerberos SAR can be more easily (and thus, more probably) secured against infrastructural attack than a more complex, less widely-implemented certificate-based SAR. As certificate-based systems come of age, however, systems administrators may find that PKI-based SAR solutions gain ground in this respect.

Like the PKI/certificate approach, the Kerberos approach to building a SAR is not without development costs. Applications must be designed or modified to support Kerberos (must be "Kerberized") in order to participate in a Kerberos-based SAR, and although a number of applications and operating systems already provide some measure of Kerberos support, many do not. Depending on an organization's installed base and usage patterns, adoption of a Kerberos-based SAR may require significantly more or significantly less retrofitting of existing applications and systems than a PKI-based SAR.

Experience at Duke, where Kerberos has been in use for a number of years in a SAR supporting a variety of public computing labs on campus, has shown that Kerberizing applications whose intrinsic authentication models are relatively modular, while not trivial, is relatively straightforward. The published Kerberos API, with its support for RFC 1510's GSS-API standard, makes Kerberizing most applications a matter of adding on the order of ten or twenty lines of C to server and client code. Kerberos extensions developed at Duke (including the Exu system co-developed by one of the authors) have simplified the integration of Kerberos authentication into a variety of systems-related tasks. It should be noted, of course, that not every application or system can be modified at every site. In the case of proprietary software and systems, the system administrator must frequently rely on cooperation from one or more vendors in order to implement support for a Kerberos- or PKI-based SAR. At Duke, the authors have been fortunate in having access to source code for some proprietary systems, and in having the support of upper management in the use of free and open software wherever possible.

Likewise, experience at Duke has shown the Kerberos model to be extremely scalable, supporting very large numbers of authenticatable entities (termed "principals") with little or no measurable degradation in performance or stability. Currently, the "ACPU.DUKE.EDU" Kerberos realm supports in excess of 37,000 users and provides secure, authenticated access to a range of applications from electronic mail to dial-up networking across a variety of computing platforms, from Macintoshes to Unix-based file-servers. This is not to say that there are not costs which increase as the user-base for a SAR grows (certainly, end-user support costs can increase dramatically) but rather that the administration of a Kerberos infrastructure supporting tens of thousands of users is

not significantly more complex or time-consuming than the administration of a similar infrastructure supporting only hundreds of users.

Many of the security vulnerabilities in Kerberos are addressed to varying extent in recent implementations of the authentication model; Kerberos V5 is significantly stronger as an authentication mechanism than Kerberos V4, and the DCE implementation of Kerberos offers even greater protections against certain forms of attack by modifying the ticket exchange protocol in some significant (if incompatible) ways. Kerberos implementations continue to evolve, with MIT and others investigating extensions to the Kerberos protocol to support more cryptographically secure authentication mechanisms and to integrate support for newer authentication approaches (including digital certificates and PKI-based authentication).

The authors believe that, for most organizations, a Kerberos-based SAR can provide more than adequate security with greater confidence and less administrative overhead than other SAR mechanisms. While PKI-based authentication mechanisms may offer advantages in the realm of electronic commerce, where users may not be known by the organizations they interact with until the need for authentication arises, they do not yet offer the proven reliability nor the existing installed base of standard implementations Kerberos boasts.

SSO Alternatives: The Dream of a Single Sign-On

Many of the SAR solutions discussed above might reasonably be viewed as also providing the benefits of a full SSO solution. In the Kerberos model, for example, a single authentication to the SAR acquires the client reusable credentials sufficient to authenticate to any Kerberized application without re-entering authenticator information. Likewise, in the PKI model, possession of a personal PK certificate may allow a user to freely authenticate to a variety of supported applications without re-entering passwords or other authentication information.

Nevertheless, in all but the most restricted of environments, it is unrealistic to expect that a SAR can provide a complete SSO solution for an organization. In the face of a heterogeneous computing environment comprising many different systems and applications, it is unlikely that a SAR can support the needs of the entire organization. Further, while a SAR may provide a basic level of SSO functionality (one authentication operation per user per work-session), it cannot usually provide the look-and-feel features demanded by many users. Systems administrators may find that providing a mechanism whereby users need only authenticate themselves once per work session is not enough to satisfy their user populations; users may demand not only strict SSO functionality but also a simplified authentication interface common to multiple environments.

Hence, system administrators charged with designing and building SSO infrastructures must frequently look beyond SAR options and investigate true SSO approaches. Common SSO approaches fall into three main categories: those which rely on an underlying SAR to facilitate authentication into multiple services at login-time, and those which rely on some centralized authenticator repository (a "key box") to provide multiple applications and systems with the "illusion" of a shared, single sign on, and hybrid solutions incorporating features of both the SAR-based and key box-based solutions.

Entry-Point Authentication: Pay No Attention to the Man Behind the Curtain

One common approach to providing SSO functionality involves the modification of common entry-point applications (login programs, etc.) to perform multiple authentication operations each time a user initiates a work session. In this approach, a number of different systems and applications sharing user information through some sort of SAR pass authentication information between one another in order to effect the appearance of a single sign-on solution.

Typical examples of this methodology include the Windows NT "layered GINA replacement" mechanism and the Kerberized Unix login mechanism.

The Windows NT user authentication mechanism (GINA) is designed in such a way that multiple different authentication modules can be invoked in succession when a user logs into an NT machine. Provided that a single common login and password are sufficient to authenticate a user to, for example, both an NT domain and a Novell NDS tree, layered GINA modules can be used to provide a sort of single sign-on appearance to the end-user.

Similarly, the MIT Kerberos distribution includes standard replacements for the normal Unix "login" program which, rather than checking user's identities against a local or distributed Unix passwd table, perform Kerberos authentication using the login and password supplied by the user. In essence, the Kerberized Unix login replacement is a sort of SSO mechanism, presenting an SSO interface for authenticating to both a Kerberos realm and an individual Unix machine.

Certainly, these methods have a place in organizational SSO planning. In order to implement a Kerberos-based SAR, for example, in a fashion palatable to general users, Kerberized entry-point applications (login programs, etc.) must be made available across as many platforms as possible. These sorts of approaches offer little beyond what can be achieved through a SAR alone, however, and do nothing to achieve a unified look-and-feel across applications or systems.

More importantly, these solutions still rely on underlying application- and system-level support for

some form of SAR, and in all but the least complicated situations, require significant re-engineering of entry-point code on multiple systems. Clearly, while the entry-point approach to single sign-on is of interest, it is not a comprehensive solution to the demand for SSO functionality.

The Key Box: SSO in a Fire Safe

Another common approach to the SSO problem, and one which seems to have been *en vogue* among vendors in recent years, is the "key box" mechanism. A central authenticator repository is established containing each user's various application- and system-specific authenticators (login/password pairs, certificates, etc.). Rather than the user authenticating to individual applications and systems directly, the user authenticates once to the central SSO server, which in turn sends the necessary authenticator information to his or her client machine to allow an SSO client application to authenticate to other systems and applications on the user's behalf. Typically, the SSO client application populates the user's graphical desktop with icons (or otherwise makes available a list of executable links) for each of the systems the user has SSO access to. Launching one of the "virtual" applications set up by the SSO client causes the appropriate application to be started and automatically authenticates the user to the application using whatever authenticator information was provided by the SSO server. In effect, the SSO client provides the user with the appearance of a single sign-on by performing authentication operations on the user's behalf.

Unlike the entry-point authentication approach, the key box approach doesn't require the prior existence of a SAR. Because the SSO server can store multiple different authenticators for a single user (one for each different system and application the user is authorized to access), there is no need for a SAR as such. Multiple authentication operations are performed during each work session, but they may be performed entirely without the user's knowledge.

While key box approaches to the SSO problem can provide a high level of functionality for end users, they can also pose some difficult challenges for system administrators, both in the realm of providing adequate security and in the realm of managing users' authentication information.

Clearly, the security the key box or SSO server system is of primary importance, since compromise of that system would lead directly to compromise of all systems and applications participating in the SSO solution. Further, ensuring the security of communications between the SSO server and its clients is of paramount importance. Depending on the specific implementation, users' authenticators may be repeatedly exposed to passive network attacks as they are transferred from the SSO server system to various client machines. Since the SSO client must typically store reusable authenticators for its users locally in

order to provide the appearance of a single sign-on environment, SSO clients must themselves be secure, lest client-based attacks permit the compromise of all of a particular user's authenticators.

Additionally, SSO clients must be specifically designed to interoperate with target applications and systems. In order for an SSO client to perform authentication on behalf of the user, it not only must have access to the user's authenticators via the SSO server, but also must have special knowledge of the native authentication interfaces used by each SSO-supported application. As applications and systems change, SSO clients may need to be modified or replaced in order to support new authentication interfaces.

Likewise, key box-based SSO solutions introduce special problems for user and authenticator management. While the key box-based SSO approach can simplify authentication for the end user, it does not by itself address the system administrator's need to reduce the number of individual authenticators which must be validated, issued, and maintained for each user. Further, the SSO software must provide some mechanism for maintaining (adding, removing, and updating) authenticators stored in the central SSO repository, and should ideally provide some mechanism for ensuring that the authenticators housed in the central SSO repository are kept synchronized with application- and system-specific authentication information. In some implementations, end-users may not be privy to their own authenticator information; login ids and passwords, for example, may be chosen and maintained by the SSO server. While this can simplify the management of a large set of users or participating systems, it can lead to serious problems if user's clients are not carefully managed. If a user's client loses the ability to interact properly with the SSO server and becomes unable to retrieve authenticators from the server, the user has no means of authenticating to other systems. In many cases, the benefits of central user management afforded by such SSO systems can be completely eroded by the associated increase in system management effort needed to ensure proper configuration of client systems.

GSO: IBM's Lock Box Solution

IBM's Global Sign-On (GSO) product is a typical implementation of the key box approach to SSO. In the GSO implementation, users' authentication information is warehoused on one or more central GSO servers, which also provide DCE-based authentication services. Users install GSO clients on their workstations which: manage the initial authentication operations necessary to access the GSO server, present a virtual desktop prepopulated with icons for all the applications for which the GSO server holds the user's authenticators, and transparently perform authentication on behalf of the user for each supported application when it is invoked. Once a user has started the GSO client and authenticated to the GSO server, the

user need not perform any further authentication operations to access supported applications. Further, the GSO server provides built-in mechanisms for automatically responding to password aging as well as a user interface for changing passwords on supported systems. Recently, IBM has begun marketing the GSO product in conjunction with Tivoli remote system management products in an effort to provide a comprehensive user management/SSO solution.

The IBM GSO product is exemplary among its competitors in its attention to security and its integration with a remote management facility (Tivoli). By relying on DCE for its primary user authentication, the GSO product can provide some of the advantages of a strong SAR in addition to providing the advantages of a targeted SSO solution. One of the strengths of DCE (and its underlying Kerberos structure) as a method of primary authentication is the natural way it provides for session-level encryption of data flowing over a network. IBM's GSO takes advantage of this strength to provide a fair level of security in the transmission of sensitive data (logins, passwords, etc.) between the GSO server and its clients.

By integrating the product with Tivoli, IBM circumvents two of the most difficult administrative problems associated with the use of targeted SSO solutions: the distribution and maintenance of SSO client software across an organization and the creation and management of system- and application-specific authentication identities across distributed systems. The inner workings of Tivoli's TME-10 product are beyond the scope of this paper, but in brief, Tivoli brings to the GSO product a mechanism for centralized installation and upgrade of GSO client software across a large organization, and provides a mechanism for creating and registering with the GSO server new user identities across a range of systems.

Similar solutions (differing mostly in the methods used to authenticate users to the central key box server and the mechanisms by which application-specific authentication operations are proxied by the SSO client) are offered by other vendors, including Platinum Technologies (in a product called Platinum AutoSecure) and CoreChange (a vendor specializing in SSO solutions for the health-care industry).

Hybrid Solutions: Best of Both Worlds

SSO solutions share a primary goal: simplifying authentication operations for the end-user. Issues of actual security and manageability are frequently of secondary importance in the design of packaged SSO solutions. At times, the system administrator is confronted with an apparent trade-off between addressing the demands of end users through an SSO solution, on the one hand, and managing and securing systems adequately through a SAR, on the other.

The authors believe that a hybrid approach to the SSO problem is often the system administrator's best

option. Such a hybrid approach would rely on a strongly-secure SAR solution to provide the appearance of a single sign-on for the most critical systems and applications within an organization and would also rely on a possibly-integrated SSO solution to support those systems which cannot feasibly be integrated into a SAR and to provide end-users with the look-and-feel features they want. By building a key box SSO solution on top of a secure SAR mechanism (such as Kerberos), the system administrator can often address organizational security and management requirements as well as end user ease-of-operation requirements with a single comprehensive solution.

SnareWorks: Hybrid Key Box and SAR

One such product, which the authors have recommended for use at Duke, is the SnareWorks product available from IntelliSoft, Inc. The SnareWorks solution sits atop a minimal DCE infrastructure (to provide security and authentication services), and provides both end-to-end encryption of network traffic and authentication/single sign-on services to supported applications.

In the SnareWorks approach, network servers are equipped with a SnareWorks server application which takes control of incoming network connections on the server. Likewise, network clients are equipped with a "thin" SnareWorks client application which takes control of outgoing network connections on the client. A centrally-managed SnareWorks "master server" manages information about the security, authentication, and single sign-on requirements of individual servers and clients, and provides for rather fine-grained definition of security rules. A particular IP port on a particular server can be configured, for example, to require authentication before accepting incoming connections, and to perform different levels of encryption at different times of day, when talking to different client machines, and when manipulating connections authenticated as different users.

When a "snared" client machine connects to a "snared" server, the SnareWorks software on the two participating machines negotiates the levels of security to be provided for the new connection: whether to perform network-level encryption, and if so, using what keys and what encryption mechanisms, whether to require authentication before establishing the connection, and what, if any, "single sign-on" services to provide over the connection. If the machines agree that authentication is required, the SnareWorks client checks to determine whether authentication credentials have already been obtained from the associated DCE cell, and if they have not, automatically prompts its user for authenticators with which to acquire credentials. If credentials are already cached within the SnareWorks client, the client engages in a normal DCE/Kerberos authentication interchange with the server to prove its identity. Depending on the configuration of the SnareWorks client and server in question,

unauthenticated connections may be refused, or may be treated differently from authenticated connections.

SnareWorks provides single sign-on features through a secure key box mechanism based on the necessarily-present DCE registry. Using extended registry attributes, the SnareWorks software stores authenticator information for various users on various systems and applications in the DCE cell's central registry. On snared servers for which single sign-on support is available, small loadable modules (termed "Program Support Modules" or PSMs) are installed which encapsulate the information necessary for the SnareWorks server to perform application- or system-level authentication on behalf of pre-authenticated users. Single sign-on is effected through the PSM when the appropriate SnareWorks server retrieves (via a DCE Secure RPC channel) authenticator information for an already-authenticated user and intercepts the authentication transaction on the server. In essence, the PSM running on the server plays the role of the client, providing a single sign-on appearance without requiring any interaction on the part of the client. SnareWorks clients provide a straightforward user interface for creating and modifying user-specific single sign-on information in an associated DCE registry, and SnareWorks PSMs provide support for server-driven authenticator management (single sign-on information updates resulting from authenticator or password expiration, for example). SnareWorks comes pre-packaged with PSMs for a number of common network-based applications and protocols (telnet, rlogin, ftp, LDAP, X11, Oracle-8, etc.) and offers a relatively simple API for constructing PSMs to support single sign-on features for more exotic or site-specific applications.

With the addition of a SnareWorks Web server interface, the SnareWorks software can be used to provide secure authentication for and highly-configurable control over web-based CGI applications running on standard HTTP servers. Unlike other SnareWorks products, the SnareWorks Web server can provide these features to end-users running on clients which do not themselves have SnareWorks client software installed – the SnareWorks Web interface only requires the existence of a web browser on client machines. With the addition of an optional SnareWorks CA, the software can be made to interoperate with certificate-based authentication and authorization strategies and can be made a part of an organizational PKI.

The SnareWorks approach offers a number of advantages both as a network security mechanism and as a single sign-on solution. Based on a secure strong authentication mechanism (DCE), SnareWorks can provide all of the features of a secure SAR for applications which have native support for Kerberos or DCE authentication. Further, SnareWorks provides a flexible, configurable mechanism for enforcing encryption of application data flowing over a possibly-insecure

network. SnareWorks also provides a scalable and retargetable mechanism for providing a single sign-on presentation to end-users, all without requiring modification of any application client or application server code. Security management is centralized through a simplified administrative user interface, allowing security rules to be changed in one location and propagated to snared servers throughout an organization. Likewise, management of the authentication information stored on an associated DCE cell is simplified through the SnareWorks administration interface which can, if so desired, be configured to provide for automatic registration of users within a DCE cell.

The approach does have some shortcomings, however. Currently, IntelliSoft offers client support for Solaris, AIX, and HP/UX workstations and Microsoft Windows 95 and Windows NT client machines. Solaris, AIX, HP/UX, and Windows NT servers are also supported. Other platforms (including Apple's Macintosh) are not supported natively, although certain features can be presented to unsupported clients through the SnareWorks web interface. Additionally, although source code to all of IntelliSoft's PSMs is available, the solution is a proprietary vended application; sites intent on having source code available locally for all critical software may find it difficult to justify this approach.

The Duke Authentication Project: A Short Case Study

As mentioned above, Duke University is currently involved in a major project to provide institution-wide authentication, security, and single sign-on services. The project was originally conceived by senior management within the University's Office of Information Technology as a means of simplifying use of certain newly-deployed enterprise-wide applications (e.g., SAP/R3, PeopleSoft, Lotus Notes). Initially, the project was directed at finding an SSO-only solution (with or without implementing a SAR) which would allow users to authenticate only once per work session, regardless of the applications they might be using.

A review committee was formed, with the authors as co-chairs and comprising representatives from major computing organizations on campus involved in enterprise-wide applications deployment, and tasked with investigating available technologies and returning an implementation proposal. Originally, this process was expected to take only two or three months, and result in a plan which could be implemented in roughly the same timeframe. After some initial discussions within the committee, it became clear that the process would be more involved than had previously been expected.

As the committee process unfolded, it became clear that different constituencies within the organization had different priorities and different requirements

for an "acceptable" solution. Those of us involved in system administration were primarily concerned with ensuring the security of the authentication service and those applications and systems it supports. Those involved in application programming were concerned with reducing the impact of any new infrastructure on existing applications and systems. End-user support staff were primarily concerned with ease of use issues, while management was primarily concerned with finding a cost-effective solution.

Starting from these different points of view, the committee were able to arrive at a number of site-specific requirements for an acceptable solution. Among these were:

- The absence of a "flag day" visible to end-users. Whatever solutions might be deployed, they should be deployed with a minimum of disruption to existing end-user work-flow patterns, and should be introduced into the existing environment in a stepwise fashion, allowing users to adjust gracefully to changing usage patterns.
- Compatibility with existing infrastructures. At the start of the investigation, Duke already had a large investment in Kerberos technology, having operated for a number of years a central Unix computing facility using Kerberos version 4 to support in excess of 35,000 users. Further, a number of applications were already in use across the enterprise, and would need to be supported by any successful SSO solution.
- The absence of any recertification of users. The University had, at the inception of this investigation, already performed significant identity verification for well-over 35,000 of its constituents in order to issue them Kerberos authenticators. The cost of this verification (which had originally been performed over a period of many years) was significant, and the committee as a whole agreed that any solution which would require the recertification of existing users (i.e., which would require issuing new authenticators to individuals already in possession of Kerberos principals and passwords) would be less than optimal.

After three months of discussion regarding the functional requirements of each constituency for an institutional authentication/single sign-on project, the committee began the process of reviewing possible solutions. The group reviewed self-maintained solutions incorporating MIT's Kerberos version 5 and various available PKI components, as well as vended solutions from IBM, Platinum, CyberSafe, Tivoli, and Transarc/IntelliSoft. Some solutions were eliminated from consideration on the basis of their not meeting the committee's requirements for scalability or supportability (CyberSafe, Platinum AutoSecure). Other solutions were eliminated on the basis of their not providing the security enhancements desired in

conjunction with deploying an enterprise-wide single sign-on solution (IBM GSO, Tivoli User Management). Ultimately, the committee agreed to recommend a solution based on IntelliSoft's SnareWorks software.

At the time of this writing, the institutional single sign-on project at Duke is completing the final stages of the organizational funding review. Once a funding model is established for the project, work can begin in earnest on the wide deployment of authentication, security, and single sign-on features called for in the committee's recommendations at the institution.

Conclusions: Learning From Duke's Experience So Far

Clearly, implementing SSO for any but the smallest and most homogeneous of enterprises is a highly organization-specific task, and certainly no one can provide a true "cook book" for arriving at an agreeable and functional solution for use in all environments. Nevertheless, the authors are convinced that as organizational computing infrastructures become increasingly complex and users become more dissatisfied with the ease of use problems associated with heterogeneous computing environments, system administrators will increasingly be called upon to provide "single sign-on" solutions for their organizations. If the question hasn't been asked yet, rest assured that it will.

While no single set of rules can guide the administrator in designing a single sign-on solution for a particular organization, a few guidelines may be of assistance in the decision-making process:

- Any complete SSO solution must embrace not only critical applications, but also the operating systems they depend upon, the network services they rely on (ftp, electronic mail, etc.), and the underlying data services (HTTP services, database management services, etc.) required by applications. An SSO solution which supports only a handful of critical applications but does not encompass the underlying services those applications rely upon will be incomplete at best, and may well fail to serve the needs of end-users, not to mention the requirements of system administrators.
- In designing an SSO solution, weigh not only the cost of deploying SSO-related software and hardware, but also the cost of supporting the envisioned authentication and/or SSO infrastructure. Pay special attention to the "hidden" costs of re-engineering applications and operating systems to support any chosen SAR or SSO solution, and to the possibly-enormous costs associated with verifying user identities and (re)issuing authenticators to end-users. Frequently, these costs will far exceed the initial capital costs of deploying an SSO solution for an organization.
- Keep in mind any security policies already in place within the organization, and those which may be mandated by any given SSO solution. Duke, in particular, has suffered from a dearth of strong security policies, so enabling enforcement of stronger security policies was a significant factor in our investigations. Sites with established policies should take care to review the effects an SSO solution may have on the enforcement of existing policies before making any sort of SSO deployment decision.
- Decide initially which of the possible key features are of the greatest importance to the organization. We recommend consultation with both end-users and management to investigate the relative importance of four key features to the organization:
 - Reduced authenticators/sign-ons. If the primary goal within the organization is to eliminate the proliferation of authenticator information for end-users, a classical, vended SSO solution may be adequate. If the primary goal is to eliminate the proliferation of authenticators from the point of view of the system administrator, a SAR may suffice.
 - Site policy control. If the primary goal within the organization is to achieve tighter control over the enforcement of site security policies, a SAR may be the best solution, possibly with a vended "key box"-style SSO solution.
 - Security enhancement. If the primary goal within the organization is to tighten security for specific applications and systems, a SAR approach using one of the more secure mechanisms discussed above may be ideal. In the case of Duke University, the additional security enhancements provided by SnareWorks' network encryption facilities were a significant factor in the choice of IntelliSoft's products. If security enhancement is of little or no concern, classical SSO solutions (such as IBM's GSO) may be most appropriate.
 - End-user convenience. If the primary goal within the organization is simplifying authentication interfaces presented to end-users, a targeted SSO solution, most likely without the deployment of a network-based SAR, may be the most appropriate solution. In these environments, organizations should consider the possibility of employing a "niche" SSO solution (CoreChange, for example, offers an SSO solution specifically targeted at the health-care market); such tightly-focused solutions can usually

provide end-users with features not available in more generic SSO solutions, but are frequently not applicable to the wide range of applications and systems for which large, unfocused organizations may need support.

- Plan from the outset for a much larger system than can possibly be envisioned. When the SSO project at Duke was first discussed, it was viewed as a relatively simple project supporting a few thousand users and a handful of mostly administrative systems. In the process of fleshing-out the details of various constituents' needs, we on the Duke project committee found that what was actually needed was an institution-wide infrastructure capable of supporting the 40,000+ current organizational affiliates, and capable of expanding to support a growing population of "new" affiliates associated with the University's ever-growing Duke Health Systems initiative. Building a system to be scalable can be difficult, but redressing a too-small solution once it has been deployed can be nearly impossible.
- Take advantage of any tools and resources already in place within the organization. At Duke, the prior existence of a large Kerberos version 4 realm has provided the institution with a "leg up" in developing an institutional SAR. While the decision to retain compatibility with the existing Kerberos version 4 realm did limit our choices with respect to SAR- and SSO-building products, it has enabled the institution to consider implementing an organization-wide solution without the initial costs associated with re-verifying 35,000 users' individual identities.
- Set reasonable, attainable goals for the SSO solution. Initial discussions at Duke centered on finding an SSO solution which would solve authentication and single sign-on problems for every application and environment in use at the institution: a full-scale single sign-on approach. Later, we came to realize that the goal of eliminating *all* user identity proliferation was too ambitious, and we have instead opted for developing a "reduced sign-on" solution. This has allowed us to provide for a higher level of functionality for those applications and systems most critical to the organization, at the expense of not supporting some less-critical applications and guaranteeing that some users of legacy applications will own multiple authenticators.
- Avoid implementation plans that introduce "flag days" for end-users. This is a lesson the authors learned in managing the growth and reorganization of institutional electronic mail infrastructures at Duke, and which applies at least as well to the deployment of SSO

solutions within organizations. Flag days are extremely expensive in terms of goodwill capital with end-users, and in the case of SSO solutions, are likely to result in the sorts of unrecoverable failures which can require an entire organization to be recertified. A gradual, incremental approach to deploying an SSO solution can prevent organization-wide catastrophes in the event that unforeseen problems arise in the deployment of an enterprise-wide authentication mechanism.

A Note on Committee Dynamics: Staying Sane in a Political Minefield

If one factor can be expected to appear ubiquitously in the quest for organizational SAR and SSO solutions, it is politics. Having worked within the Duke computing infrastructure for a combined total of over 17 years, the authors expected political factors to interfere with the progress of Duke's institutional SSO project, yet even we were somewhat surprised at the extent to which the "SSO issue" was cause for political intrigue. Here are a few tips derived from our experience with the Duke SSO review committee:

- Take the time to involve a diverse cross-section of the community throughout the decision-making process.

This was perhaps the most difficult and the most useful approach we used. We were fortunate at Duke, in that the request for a "single sign-on solution" came from relatively high on the corporate ladder (the Office of the Associate CIO). This facilitated our enlisting other computing professionals from within the central computing support groups at the University, as well as end-users, administrators, and managers from various departments around campus in our work. Although the diversity of the committee made our initial work slow (some members of the committee started with virtually no understanding of the issues involved), it paid off in the end with a broad base of support for the project, once it was fully described.

- Take the time to address first principles, and find common goals across the participants.

In our case, this was a significant challenge. In particular, the authors found that with the exception of a few extremely technical staff virtually no one on the SSO review committee came into the process with a clear understanding of what "SSO" really meant. Many hours of meetings were necessary just to clarify the distinction between *authentication* and *authorization* for the committee (most members failed to see the distinction at first), and many more hours were necessary to work through the trade-offs between ease-of-use factors (of critical importance to end-users and managers) and security factors (of critical importance to

system administrators and programmers). In the end, more than half of the time spent in committee was spent discussing "first principles." Once these issues were agreed upon, however, vendor reviews and the choice of an approach were comparatively simple.

- Expect the process to take some time.

At Duke, our original mandate was to complete the review of available options and return to the senior administration a proposal within three months. At the three-month mark, our committee had barely achieved agreement on the nature of the problem at hand, let alone selected a particular solution. The unfortunate absence of the authors' departmental director as a result of an extended illness allowed us to petition the administration for additional time. However, pressure mounted as scheduling of vendor presentations and meetings with the review committee and key managers outside the institutional computing infrastructure forced the project schedule to slip further and further.

- Expect the process to be expensive. Warn your management early.

No matter what path an organization takes through the SAR/SSO obstacle course, be assured that the result will be more expensive than originally expected. If the solution(s) chosen are proprietary in nature, much of the expense may be in software licensing and acquisition. If the solution(s) chosen are "free," the costs may be weighted toward staff time for development and installation. Regardless, we urge system administrators working on SSO solutions for their organizations to be open and up-front with management about the real costs involved. At Duke, we expect the initial year of development and deployment to cost in the neighborhood of \$1 million, with continuing costs between \$100,000 and \$200,000 per year. Original estimates of the total cost for an institutional SSO project were significantly lower, although compared to the costs of some of the major efforts ongoing at the institution (some of which will run into the tens of millions of dollars), the expense is less staggering. Note that Duke is a large and diverse enterprise, with more than 40,000 individual users. Your mileage may vary.

Software Availability

See Table 4 for URL references for software discussed in this paper.

Author Information

Michael Fleming Grubb has worked as a Unix system administrator at Duke University for six years. He is the principal architect of the campus-wide email and web infrastructure. He is also a licensed attorney.

He can be reached via post at Box 90132, Duke University, Durham, NC 27708-0132, USA, or via email at <mg@duke.edu>.

Rob Carter has worked as a Unix system administrator at Duke University for over eleven years. He has been the lead system administrator for the University's public Unix computing facilities throughout their existence, although he did not invent Unix. He can be reached via post at Box 90132, Duke University, Durham, NC 27708-0132, USA, or via email at <rob@duke.edu>.

References

- Bellovin, S., and M. Merritt. "Limitations of the Kerberos Authentication System." *ACM Computer Communications Review*, October 1990.
- Dyer, S. "The Hesiod Name Server." *USENIX Conference Proceedings*, Winter 1988.
- Fraser, B., ed. *RFC 2196: Site Security Handbook*, 1997.
- Haller, N. and R. Atkinson. *RFC 1704: On Internet Authentication*, 1994.
- Haller, N., "The S/Key One-time Password System." *Proceedings of the Symposium on Network & Distributed Systems Security*, Internet Society, San Diego, CA, February 1994.
- Haller, N., C. Metz, P. Nesser, and M. Straw. *RFC 2289: A One-Time Password System*, 1998.
- Hess, D., D. Safford, and U. Pooch. "A Unix Network Protocol Security Study: Network Information Service," *ACM Computer Communications Review* 22 (5), 1992.
- Kaufman, C., R. Perlman, and M. Speciner. *Network Security: Private Communication in a Public World*. Prentice Hall, 1995.
- Kohl, J., and C. Neuman. *RFC 1510: The Kerberos Network Authentication Service (V5)*, 1993.
- Linn, J. *RFC 1964: The Kerberos Version 5 GSS-API Mechanism*, 1996.
- Linn, J. *RFC 2078: General Security Service Application Program Interface*, Version 2, 1997.
- Mockapetris, P. *RFC 1034: Domain Names - Concepts and Facilities*, 1987.
- Mockapetris, P. *RFC 1035: Domain Names - Implementation and Specification*, 1987.
- Mullan, S. *OSF RFC 92.0: DCE Interoperability with Kerberos*, 1996.
- Pato, J. *OSF RFC 6.0: A Generic Interface for Extended Registry Attributes*, 1992.
- Ramm, K. and M. Grubb. "Exu - A System for Secure Delegation of Authority on an Insecure Network." *Proceedings of the Ninth Systems Administration Conference (LISA IX)*, Monterey, CA, September 1995.
- Ramsey, R. *All About Administering NIS+*. SunSoft Press, 1994.
- Rigney, C., A. Rubens, W. Simpson, and S. Willens.

RFC 2138: Remote Authentication Dial In User Service (RADIUS), 1997.

Steiner, J., C. Neuman, and J. Schiller. "Kerberos: An Authentication Service for Open Network Systems." *USENIX Conference Proceedings*, Winter 1988.

Stern, Hal. *Managing NFS and NIS*. O'Reilly & Assocs., 1991.

Wray, J. *OSF RFC 5.2: GSS-API Extensions for DCE*, 1994.

Appendix 1: Tables

Application or System	Short-Term	Medium-Term	Long-Term
AFS	X	X	X
Unix net apps (ftp, telnet, etc.)	X	X	X
X11	X	X	X
IMAP4	X	X	X
POP3	X	X	X
HTTP/CGI	X	X	X
SAP R/3	X	X	
PeopleSoft	X	X	
LDAP	X	X	
Lotus Notes	X	X	
MVS/RACF	X		
Oracle8	X		
Netware (4+)	X		

Table 1: Systems and Applications Targeted for SAR/SSO at Duke.

Feature	Critical	Important	Nice-to-have
Strong Authentication	X		
Distributable Management	X		
Unix compatibility	X		
NT compatibility	X		
Standards Compliance	X		
Scalability	X		
Performance	X		
Network Security	X		
Stability/Easy maint.	X		
Win95/Win98 compat.		X	
MacOS support		X	
SSO features		X	
Application Support		X	
Published API		X	
SSO user interface			X
Automated user registration			X
Remote installation			X

Table 2: Decision Matrix: Features Required for SAR/SSO at Duke.

Vendor	Product	Strng Auth	Dstrb Mgmt	Unix Supp	NT Supp	Stds Compl	Scala bility	Perf .	Net Sec	Stab lity	Win 95	Mac OS
MIT	Hesiod	-	=	+	-	=	=	+	-	=	-	-
Sun	NIS	-	=	+	-	=	-	+	-	-	-	-
Sun	NIS+	-	+	=	-	-	=	=	=	-	-	-
---	S/Key	=	-	=	-	-	-	=	+	=	-	-
MIT	K4	+	=	+	=	+	+	+	=	+	=	=
MIT	K5	+	=	+	=	+	+	+	+	+	=	=
CyberSafe	---	+	=	+	=	=	+	?	+	=	=	-
IBM	GSO	=	+	=	+	=	+	?	=	+	+	-
Platinum	AutoSecure	-	+	=	+	-	=	?	-	?	+	-
IntelliSoft	SnareWorks	+	+	+	+	+	+	+	+	=	+	-

Vendor	Product	SSO features	Applic. Support	Pub API	SSO-GUI	User Regis.	Install/ Maint.
MIT	Hesiod	=	= (+/Unix)	+	-	-	=
Sun	NIS	=	= (+/Unix)	=	-	-	=
Sun	NIS+	=	- (+/Solaris)	-	-	-	-
---	S/Key	-	=	=	-	-	-
MIT	K4	+	=	+	-	-	=
MIT	K5	+	-	+	-	-	=
CyberSafe	----	+	=	-	=	-	?
IBM	GSO	+	+	-	+	=	+
Platinum	AutoSecure	+	=	-	+	=	?
IntelliSoft	SnareWorks	+	=	+	=	+	+

KEY

- +: strength of product
- =: available, but not particular product strength
- : not available or too weak to rely upon
- ?: insufficient data gathered to determine

Table 3: Vendor Performance: Features of Vended Products.

Product	Location
Hesiod	ftp://athena-dist.mit.edu/pub/ATHENA/hesiod/
NIS	(Distributed with most Unix variants)
NIS+	http://www.sun.com/software/white-papers/wp-nisplus/
RADIUS	http://www.livingston.com/tech/docs/radius/
S/Key	ftp://ftp.bellcore.com/pub/nmh/
SecurID	http://www.securid.com/
MIT Kerberos V4	ftp://athena-dist.mit.edu/pub/kerberos/README.KRB4
MIT Kerberos V5	http://web.mit.edu/kerberos/www/
DCE	http://www.opengroup.org/dce/
KTH Kerberos V4	http://www.pdc.kth.se/kth-krb/
Tcl-Kerberos	http://www.neosoft.com/tcl/ftparchive/sorted/net/tcl-krb5/
Java-Kerberos	http://www.camb.opengroup.org/RI/www/jkrb/
GSO	http://www.software.ibm.com/enetwork/global/signon
SnareWorks	http://www.isoft.com/
CoreChange	http://www.corechange.com/
Platinum AutoSecure	http://www.platinum.com/products/sysman/security.htm
Tivoli	http://www.tivoli.com/

Table 4: Software Availability.

Protocol	PSM	Authentication	Encryption	Authorization	SSO
Telnet, FTP	Yes	Yes	Yes	Yes	Yes
X11	Yes	Yes	Yes	Yes	N/A
SMTP	Yes	Yes	Yes	Yes	N/A
IMAP4	Yes	Yes	Yes	Yes	Yes
HTTP	Yes	Yes	Yes	Yes	Yes
POP3	Yes	Yes	Yes	Yes	Yes
R{login,sh}	Yes	Yes	Yes	Yes	Yes
NNTP	Yes	Yes	Yes	Yes	Yes
IOPNS	Yes	Yes	Yes	Yes	Yes
SNMP	Yes	Yes	Yes	Yes	Yes
MQseries	Yes	Yes	Yes	Yes	Yes
PeopleSoft	Devel	Yes	Yes	Devel	Devel
SAP	Devel	Yes	Yes	Devel	Devel
LDAP	Yes	Yes	Yes	Yes	Yes
Lotus Notes	Devel*	Yes	Yes	Devel*	Devel*
SMB/Netbios	Yes	Yes	Yes	Yes	Yes
Oracle8	Yes	Yes	Yes	Yes	Yes
Arbitrary IP protocols	No**	Yes	Yes	No**	No**

KEY

Yes:	SnareWorks provides this feature for the protocol
Devel:	SnareWorks is currently developing this feature
No:	SnareWorks does not provide this feature

NOTES

- * Upcoming versions of Notes are reported to support X.509 certificate-based authentication. Such would make a Notes PSM redundant, since SnareWorks provides native support for X.509.
- ** The SnareWorks API provides a mechanism for sites to develop their own PSMs, providing SSO and Authorization features for arbitrary IP protocols.

Table 5: SnareWorks Application Support Levels.

Using Gigabit Ethernet to Backup Six Terabytes

W. Curtis Preston – Hughes Space and Communications

ABSTRACT

Imagine having to prove everything you believe at one time. That is exactly what happened when I was asked to change the design of an Enterprise Backup System to accommodate the backup and restore needs of a new, very large system. To meet the challenge, I'd have to use three brand-new pieces of technology and push my chosen backup software to its limits. Would I be able to send that much data over the network to a central location? Would I be forced to change my design? This paper is the story of the Proof of Concept Test that answered these questions.

The Challenge

During the design of an enterprise-wide backup system it was announced that the client was purchasing a Sun Enterprise 10000 with six terabytes of data, and an EMC three-terabyte storage array. The E-10000 would provide database, application and file server support. The EMC array would initially be attached to an NFS file server with one terabyte of exported data. Systems of this size would place significant new demands on the backup and recovery system. We needed a much more powerful system than we had initially envisioned!

The Answer

The capabilities of Gigabit Ethernet, Sun's E-450, Quantum's DLT 7000, and Legato's NetWorker allowed us to accommodate this large server by changing only a few components of the original design. The current system design backs up data at speeds of up to 144 GB/hr. This means that we can back up a 1.1TB system (or E-10000 domain) across the network within an 8-hour backup window.

How does one get a terabyte of data to a tape on the other side of a network within eight hours? The answer is dynamic parallelism. Although several commercial products support this when backing up databases, only three support backing up file systems in this way. (For details on the concept of dynamic parallelism, please refer to <http://www.backupcentral.com/parallel.html>.) Legato NetWorker was already chosen based on a number of factors, and it supports this kind of backup.

Once we realized the amount of data that would be sent to the backup server, we began to wonder about its capabilities as well. Just how much data can you push through an E-450? The answer, thankfully, was "a lot!" The system is specifically designed for I/O.

Assuming the software can transfer the data and the backup server can accept the data at the needed

rate, it needs somewhere to put it as well. DLT 7000s were chosen due to their price/performance ratio, as well as their reliability record with this company.

Since we had to backup terabytes of data across the network, we were going to need a pretty big network! The good news is that Gigabit Ethernet cards and switches just began shipping a few months ago. Our design, originally based on 100BaseT, was quickly changed to accommodate this new technology.

Each system to be backed up by our enterprise backup system is now plugged into a 10/100 or Gigabit Ethernet port on the switch, based on the required throughput for that host. This formed a completely separate backup network, with no route to the backbone.

The end result is a multi-host, multi-platform, enterprise-wide backup solution that uses a dedicated network for all backup data traffic. It is capable of backing up over 1.1 TB in one night. If we need to increase that to several terabytes, all we have to do is use more storage nodes – since the switch is capable of much more than we are currently doing with it.

History

While consulting at VBC¹, I discovered that they were trying to design a backup system for their entire enterprise. Since this is my area of specialty, I volunteered to be on the project team. The project that followed proved to be one of the most interesting ones to which I've ever been assigned. The first phase was an extensive survey of the user community. Since the client is a very big company, this took quite a while. We found two distinct philosophies within the company.

¹VBC is an acronym that I will use to refer to the client where all this work was done. While consent was given to write this paper, I find it best to keep them anonymous. In case you are wondering, VBC is short for "Very Big Client."

The network administration group felt that the current network infrastructure was grossly inadequate to handle any sort of network-based backups. They were correct. Some of their servers were still on shared 10BaseT, which can only handle about 300-400 KB/s on a good day! This group felt that the best way to do backups was to do them all to locally attached tape drives on each server.

The systems administration group felt that tape drives put servers at risk. They were also right! Backup processes can sometimes hang on a given tape drive, requiring a reboot of the server to fix them. The older the servers and tapes drives are, the more likely this is to happen. Doing all backups locally would certainly increase the chance that a production server would need to be rebooted before its backups would work again.

I was also told that upgrading the network infrastructure wasn't going to happen any time soon. I was also told that installing private networks was all right – if I made sure there was no routing from the private network to the public network.

Network Layout

The system we were to design was to handle the backups of several buildings within a five-kilometer radius (see Figure 1). There were two main buildings where most of the systems resided. I'll call them buildings "A" and "B." All buildings are currently connected via a FDDI backbone, but the network within each building is often limited to shared or switched 10BaseT and shared 100BaseT. Building A has approximately 50 servers (mostly Suns) ranging from 1-60 GB. Building B consists of approximately ten servers (mostly Suns) ranging from 1-60 GB – and one 500 GB Auspex. There is also a 500 GB Auspex

in Building C, and five buildings with less than 60 GB of data each.

While most of these servers are Solaris systems, there are a few HP-UX systems and several NT and Novell systems throughout the campus. The Novell and NT systems were not included in the scope of this project. However, we were asked to keep them in mind for future inclusion.

Design Options

There were four primary options that we considered when designing the backup system: using all local devices, using the backbone, creating a local/backbone hybrid, and a private backup network. There is also a new option that was not available during phase one, but it shows promise and will therefore be included in this report.

Backups via Local Devices

This is the traditional method for performing backups. An administrator loads a tape in a local device and some homegrown shell script magically gets the data to tape. As discussed earlier, though, it puts the servers at risk of being shut down to fix tape drive problems. It could be argued this is less likely to happen with modern systems and equipment. It is still more likely to happen, though, if you use the tape drives on the servers for backups. In today's world of commercial backup software, this also becomes a very expensive option. That is because when you use server A's tape drive to backup server B, server A is referred to as a "media server." Most backup software packages charge extra for each media server. This means that it could cost you \$5000 or more to use your \$2000 tape drive! It's also the least manageable of the options, since you would have to send someone to change tapes all over the place. Installing auto-changers everywhere to fix that would increase the

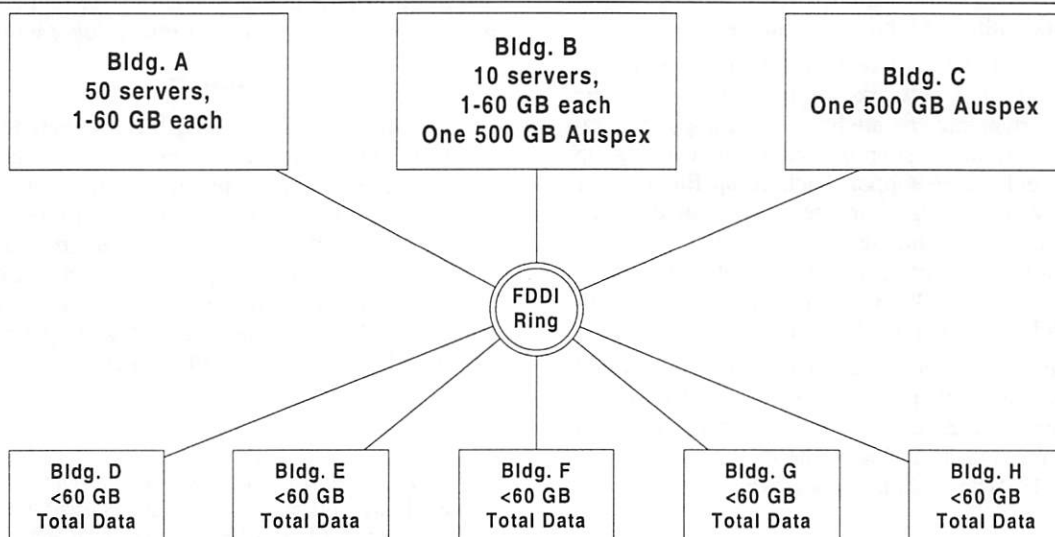


Figure 1: Network layout, 5000-ft. view.

cost astronomically due to licensing – another \$2000-\$5000 per auto-changer.

Backups via the Backbone

This is how most backup software packages started. You install a central tape library and backup everything via the backbone. This works fine if you have a large backbone. While VBC's backbone was fine, many servers had only a 10BaseT to that backbone – often shared 10BaseT. This option was quickly ruled out as being impossible.

Local/Backbone Hybrid

Can't we all just work together? What if we put tape drives on the really big servers, and backup the smaller clients across the backbone to those servers? This is used quite a lot in companies across the world because it allows for a great deal of flexibility. However, it has "hidden" problems. First, consider all the problems listed under "Backups via Local Devices," and the small amount of available network bandwidth. Even if we had more bandwidth at our disposal, we will soon outgrow it.

However, the most important problem with this design is one that does not immediately come to mind. Since you are attaching the devices to your "biggest" servers, you are probably also attaching them to your most important servers. Not only are you putting servers at risk, you are putting your most important servers at risk. Also, these servers were designed to do the job they were made to do – not to do backups for you! Causing each large server to also become a backup server puts additional load on your most important servers. This additional load will mean slower response times for the customers trying to use these machines.

Another problem with this design is the load created by restores. Many people design their backup networks without thinking about restores. Suppose for a minute that your network and servers are very slow at night while your backups are running. Most of the issues above do not apply to you. Now ask yourself what will happen when one of the servers dies during business hours? Your production server is doubling as a backup server, so it will have to slow down to respond to your restore request. Your network will come to a screeching halt while you transfer data from the backup server to the server that needs to be restored.

This option was rejected for the reasons listed above.

Private Backup Networks

What if you created a special network just for backup traffic? You would then be allowed to backup and restore any time you want. You could put the backup devices anywhere you want, and the only load on a server would be to put its own data on that network. (You will see later that this is actually the

design's only limitation.) In our particular environment, this would mean installing a separate network in each building, sizing each network by how many servers are present. Systems in each building would then backup to a local "media server" within that building, keeping all backup traffic off of the backbone.

Storage Area Networks

This is "bleeding edge" stuff, but promises to bring great things in our future. The one limitation of the "private backup network" method is that each server is responsible for transferring its data to the backup server across the private network. What if you have a very small server (e.g., an Ultra 2) that happens to have terabytes (or even petabytes) of data on-line? A good example of such a server would be an imaging server. It may contain thousands of images, but only a few of them are asked for at any one time. Its primary job is to keep track of these images. Because of this, the computer in front of the imaging system does not have to be extremely powerful. It just needs to be able to address all the storage. The end result is that you have a tremendous amount of data sitting behind an extremely small server. That server might not be powerful enough to transfer all that data over the network, no matter what kind of bandwidth it has available.

The answer is a storage area network, or SAN. It is beyond the scope of this paper, and we did not choose it because it was too bleeding edge at the time. Imagine if you had a separate, SCSI based network that contained only your storage devices. This SAN would have at its center a "SCSI switch," connecting a central storage device (such as a large RAID array) to the systems that needed to access that storage. Every system can see the disks as being locally attached and can access them at Ultra SCSI (40 MB/s) or Fibre-channel (100MB/s) speeds. A backup server could also be connected to the SAN and access every servers disk drives locally. That way it could backup a client's disks without having to use the client's CPU at all. The only problem with doing it this way is that you have one system writing to the disk and another one reading from it. The administrative overhead of resolving these issues caused us to reject this method for now. It may be more viable in the future as these issues are resolved.

Another way to use a SAN is to locally attach a large storage device (such as a tape library) to every server that needs it. This is the way that Legato's new "Smart Media" program works. It cooperates with the SCSI switch to dynamically allocate the entire library to every client while that client is trying to backup. This would work very well with the E-10000 where you have both a large server and a lot of storage attached to it. This would allow you to use the shared memory type of backup that greatly reduces your CPU load while still allowing shared access to the tape library! The only problem with this method is that it

was unavailable when we started the project! It was released September 1, 1998. It promises to be quite interesting.

Chosen Design

We decided to go with the private backup network design. It is simple, and allows for a great deal of flexibility. New clients can simply be plugged into the network and added to the configuration. They will not need to have tape drives installed on them – only the backup software. Backups and restores can happen any time without impacting anyone but the servers that are being backed up or restored. Since switches are so inexpensive now, it also turns out to be the least expensive option. VBC also has over 100 of Sun's combination SCSI/10BaseT cards in use, which means that most of the servers will not even need to buy a NIC to be added to the backup network. (This is probably the case for many Sun shops, since these cards have been widely sold over the past few years.) Logically, the network will look like the one in Figure 2.

Two New Challenges

For a lot of reasons external to the project, significant time passed between the preliminary design phase and any sort of design testing, pilot or implementation. During that time, two very large projects were introduced that would require a significantly more powerful backup system than the one described above. However, you will see that we actually decided only to change certain pieces of the design. The actual overall logical design stayed the same.

As you read this section, please do not think that it does not apply to you because you are not going to purchase an EMC array or a Sun E-10000. The problems apply to any shop with a single server that has close to (or more than) a terabyte of data. Backup software does a great job of backing up hundreds (or even thousands) of small or mid-size computers to a central location. However, when one single host becomes very large, the challenge becomes quite different.

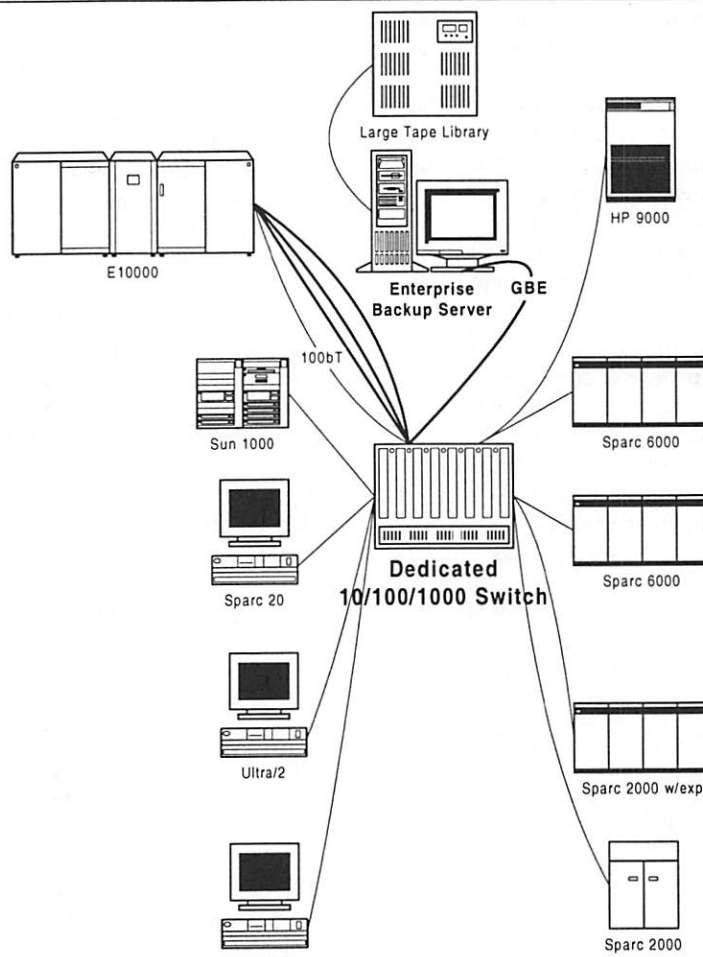


Figure 2: Dedicated backup network design.

Sun Enterprise 10000: A System That Doesn't Share

The Sun Enterprise 10000 is the result of some technology purchased from Cray when they merged with SGI. It is a very large, multi-processor, multi-host, single-backplane system. It consists of a central unit with a backplane in the center of the unit, which is why Sun calls it a "centerplane," which can accept up to 16 I/O boards. Each I/O board can handle up to four CPUs, 4 GBs of RAM, and 4 SBUS cards. That means that the system supports up to 64 CPUs, 64 GBs of RAM, and 64 SBUS cards!

The E-10000 can then create up to 8 "virtual hosts," (referred to as domains) each of which has their own boot drives and runs their own copy of the operating system. You can then dynamically allocate the I/O boards to these domains, while they are currently up and running. This allows you to allocate more CPU and memory resources to an important, cyclical task, such as end of the month (or year) accounting. Once those resources are no longer needed, you can then deallocate those resources and allocate them for some other purpose.

Although this sounds like a beautiful system, the problem it presented to us was simple matter of arithmetic. VBC was planning on having anywhere from 250 GB to 1.5 TB of storage connected to each of these domains. If we used the current backup network design, the biggest pipe we would have available would be 100BaseT. At maximum speed, a 100BaseT client can back up 288 GB in one night. (That's represents saturating the 100 Mb/s pipe. $10 \text{ MB/s} * 60 \text{ seconds} * 60 \text{ minutes} * 8 \text{ hours} = 288 \text{ GB}$) We would need a system that was approximately six times faster than that!

The E-10000 does offer other alternatives, which are similar to the others discussed earlier – with a slight twist. Remember that each domain is truly a separate host, and that means that you cannot transfer data via the centerplane from one domain to the other. You have to use the network like anybody else. However, there are some interesting options that are presented because of the unique architecture of the E-10000.

The first option is that one could mount a large jukebox off of one of the I/O boards and dynamically allocate it to each domain during the backup cycle. This would require extensive use of the Dynamic Reallocation feature of the E-10000, which is obviously completely new and very "bleeding edge." It would also present some interesting logistical challenges for the backup software. Imagine a backup jukebox that dynamically disappears and reappears on different media servers! How would you handle the inventory of such a beast? For this, and other reasons we rejected this idea.

There is also one other really new option if you are using Legato NetWorker. They have a new module referred to as "Smart Media." It allows you to connect several hosts to one library via a SAN switch (discussed above). These multiple hosts could, of course, be multiple domains within an E-10000. Legato would understand that all of these hosts have full access to the same library and dynamically handle the reassignment of it to the different hosts who need it. Only one host will be able to use the library at one time, however. This option was not available when we started the project, and we also believed that we would like to keep devices off the servers if possible.

And one final option should also be considered if you have purchased an E-10000. The current revision of the E-10000 does not support data transfer between domains, except by using the network like any other machine. Future revisions may allow this. This means that you could have one domain act as a media server, and have the other domains send their data via the centerplane to that domain. This option is currently unavailable and will have its own unique set of problems, but it's something to consider in the future.

Terabyte NFS Server

The second challenge that came about was an idea to create an NFS server that would service multiple departments, totaling about one terabyte of data hanging off of a single system. This was actually a worse problem than the one created by the E-10000. The reason is that our design called for using each client's CPU to transfer the data to the backup server via the dedicated backup network. Based on some initial tests, we were using every bit of the E-10000's resources to transfer data at gigabit speeds. The problem with this configuration is that the system that was going to drive this terabyte file server was only an Ultra 2 with two CPUs and 512 MB of RAM. This is about one eighth of what we were using on the E-10000! I believe that this system will require us to split its full backups across multiple nights. For a discussion of why we were trying not to do that, see "Single Client Backup Throughput."

New Limitations

Single Client Backup Throughput

It is important that we be able to do a full backup of any given client in one night. The reason is a feature in NetWorker called the "All Saveset." Using the All saveset automatically backs up any file systems present on the client. It requires zero administration once you configure NetWorker to use the All saveset, since it will automatically discover that you've installed a new file system and back it up. However, the one limitation of the All saveset is that you must back up the entire client each time you run a backup. This is not a problem on the nights when you are performing an incremental backup, but when it's

time for a full backup you need to be able to fit it into a single night. Splitting up a client's full backups across multiple nights means not using the All saveset and manually specifying the file systems to back up. This requires that you then monitor the client for any new file systems as they are created.

The original design was based on 100BaseT, allowing a maximum transfer rate of about 10 MB/s, which allows us to perform a full backup of a 288 GB client in an 8 hour period ($10 * 60 * 60 * 8$). Please note that both the terabyte file server and at least one domain on the E-10000 are approximately four times that size. 100BaseT was not going to do the job.

Backup Server Throughput

The backup server would obviously also need to be able to accept more than 10 MB/s of data, allowing it to back up more than 288 GB per night, or we'd never get the job done! We would need a more powerful network, and a more powerful backup server to

connect to it. One potential solution would be to install a second backup server. Another solution would be to increase the pipe.

Storage Capacity

Storage capacity would also have to be drastically increased to accommodate the needs of these new very large clients. The original jukeboxes were going to hold somewhere around 52 tapes. That would barely provide enough capacity to backup these new systems one time!

Final Design

We wanted to stay with the same logical design, because we believed that keeping tape drives off the servers was a good idea. So we decided to just beef up the subsystems of the original design. 100BaseT became Gigabit Ethernet. The Sun Ultra-2 became a Sun E-450. And the tape library went from 52 tapes and four drives to 496 tapes and eight drives! What

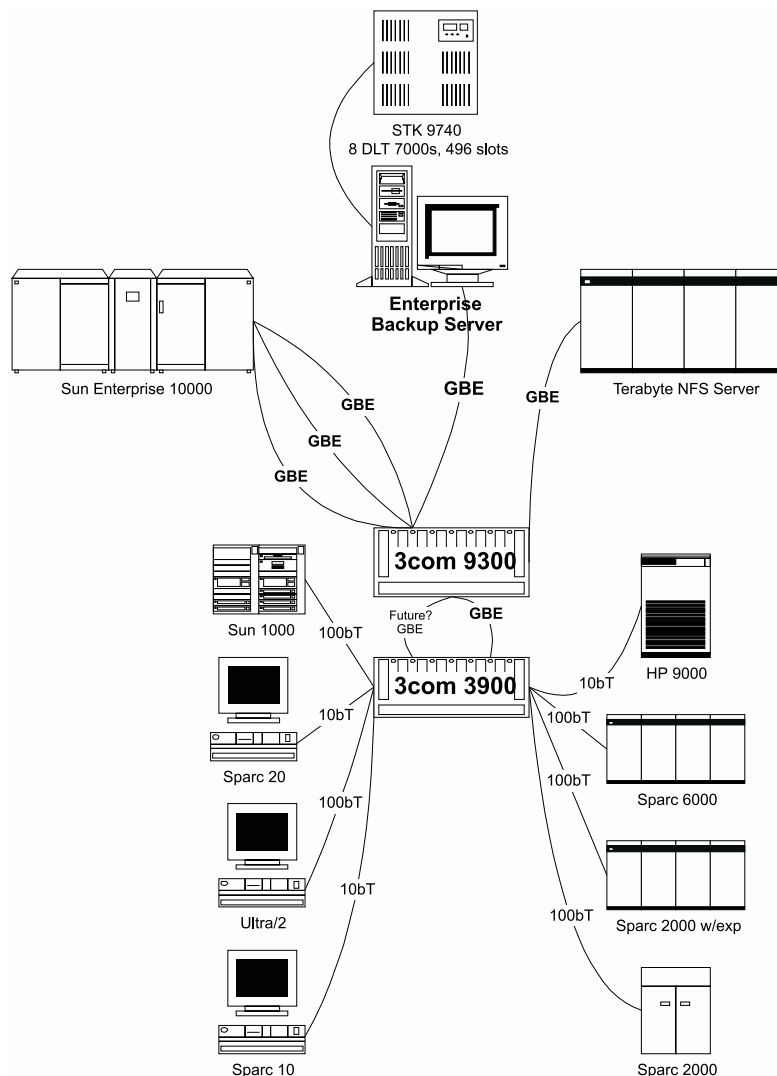


Figure 3: The final configuration.

we didn't know until we tested it, though, was whether or not we could scale this far with Legato NetWorker, and whether or not Gigabit Ethernet would actually be that much faster. First, here are some details on the pieces of the new design.

We went for a combined approach on the network. We would provide 10/100 to those clients that needed it, which would be the bulk of them. We would then hook that switch to a Gigabit Ethernet switch via a Gigabit Ethernet uplink. The switches that we chose to do this were the 3com 3900 and 9300. The 3900 has 36 10/100 ports and from one to three Gigabit Ethernet links. See Figure 3.

How Well Did it Work?

It worked well enough to get the job done, but not as well as I had hoped. My current opinion is that Gigabit Ethernet may be ready for switch-to-switch communication (and it is), but the servers are just not ready to push that amount of data out an Ethernet port – given Ethernet's inherent limitations. You can actually get about the same throughput with a quad fast Ethernet card and Cisco's Etherchannel or 3com Trunking software.

I will say a few things in Gigabit Ethernet's defense. First, I believe that it really caught the OS vendors by surprise. I had always heard of Gigabit Ethernet as a backbone technology, not something to be directly connected to a server. Who would have thought that crazy people like me would want to actually use it to sustain a transfer rate of 500 Mb/s while backing up a server? It's also important to note that the standard wasn't really ratified until just a few months ago. Also, the main Achilles' heel of Ethernet (when running at Gigabit speeds) seems to be its 1500 byte frame size. A frame size of 9000 bytes is much better suited for the task and NICs and switches from

Alteon support this frame size. They've also proven that by using that frame size they can more than double the throughput from a single server while simultaneously reducing the CPU load by 50%! So I wouldn't give up on Gigabit Ethernet just yet.

The Numbers

Since the ultimate goal of this private network was to backup the data using Legato NetWorker, most of the benchmarks were made with that same software. After getting less than optimal results, we did some other tests without it to see if it was causing the problem. The main system that we used for testing was an E-4000 with 8 CPUs and 1.5 GB of RAM. The disks we were reading from were in multiple Sun A-5000s striped for optimum speed using Veritas volume manager.

The first tests were done with NetWorker itself. By using different combinations of client, device and server parallelism, we tried to stream as many drives as possible. (We defined streaming as continuous operation at close to 10 MB/s.). Since not streaming a drive is neither good for the drive or the tape, our goal was to find the smallest number of tape drives that could achieve the highest overall throughput.

The next test was designed to determine the maximum throughput of Gigabit Ethernet. We did this by creating a file that was big enough to be significant, but small enough to fit into RAM. We then read that file into memory by copying it to /dev/null. Then we used rcp to copy it from one system to another system's /dev/null. This would result in minimal disk operations and hopefully the fastest overall throughput.

We also simulated NetWorker by issuing multiple rcp commands simultaneously, copying several large files from one system to another system's disk.

Tool	Number of Devices written to	Threads from client	Total throughput	Average throughput per device	CPU Util On client	Percentage Memory Used
NetWorker File System Backup	1	4	10_MB/s	10_MB/s	40	30
"	2	8	20_MB/s	10_MB/s	80	80
:	3	12	25_MB/s	8_MB/s	100	100+
Copy of file in memory	N/A	1	15_MB/s	N/A	10	50
"	N/A	2	30_MB/s	N/A	60	70
Multiple rcps	N/A	5	40_MB/s	N/A	80	100+
"	N/A	6	39_MB/s	N/A	90	100+
Multiple tars to tape	4	4	20_MB/s	5_MB/s	90	100+

Table 1: Backup throughput.

We then used multiple simultaneous tars to get even closer to what NetWorker was trying to do. The results of these tests are in Table 1.

There were many tests performed at different times with different amounts of memory, etc. This table represents a summary of the results we found. Pushing a Gigabit Ethernet connection at anything even close to 40 MB/s (320 Mb/s) required every bit of memory that we had. At first, we thought that it was also a CPU limitation, until we ran the same tests on a much smaller system – an Ultra 2 with 1 GB of RAM. The Ultra2, with about 25% of the CPU power that the E-4000 had, ran out of steam at about the same place. We believed, therefore, that the high CPU utilization was due to a very excessive scan rate. The CPU was spending all its time freeing up memory and very little time doing the actual job we asked it to do!

We discussed using 3 or 4 GB of RAM to see if we might have gone a little faster, but we didn't for two reasons. The first was that this was not going to be the configuration of the E-10000. The second was that we didn't have it!

Implementation Issues

This project is a large one, and we are currently in the Pilot phase. The goal is to back up one of several buildings with this configuration, and then to roll out this design to other buildings. Now that testing is out of the way and we know the capabilities and limitations of this system, we can devote time to resolving implementation issues.

Disaster Recovery

The system is setup to perform a full backup once a month and a level 5 backup once a week of every system. The NetWorker indexes also receive a full backup every day to one tape. These backups are then automatically cloned (copied) and sent offsite as soon as they are made. This offsite storage of essential data will form the basis of the disaster recovery plan.

Currently, level one and level two disasters are going to be handled by using servers in other buildings to recover essential data. There is another project to incorporate a disaster recovery service in the future. We felt that establishing a solid foundation of backups being sent off site was the first step towards making such a service valuable.

One type of small disaster that is not addressed in the pilot (but is addressed in the overall design) is the failure of the tape library. A second library in another building will be added in phase two, and it will become a storage node (remote device server) for the server in this building. If this jukebox fails, that jukebox will automatically take over!

Media Management

Several systems have been put into place to automatically:

1. Clone certain types of backups
2. Import and export the appropriate tapes to and from tape library
3. Separate full and incremental backups to allow separate retention periods of those types of backups

There are also procedures that dictate daily operations to ensure that enough tapes are available to the library at all times, as well as how to load special tapes for a restore from old tapes. All tapes that are not stored in the library will be stored in Wrightline media cabinets, customized for the types of media we have. There will also be a media database that will track the location of all tapes at all times. A dedicated operator staff will handle the physical management of all tapes.

Report/Error Notification

There are mail aliases already defined for backup success/failure notification, and this system will use those same aliases. Every backup sends an email message with the subject line of "Successful backup of \$GROUPNAME on \$HOSTNAME." If any failures are detected, this subject line is changed to "FAILED backup of \$GROUPNAME on \$HOSTNAME." This allows those reading the mails to easily notice which backups had problems.

The NetWorker bootstrap report is handled in a special way. It is:

1. Saved into a log file in /nsr/reports
2. Printed to a special printer with a header specifying to have it given to the appropriate SA
3. Emailed to the backup notification list
4. Emailed to a few external email addresses so that it will be available even after a level two disaster

Index Management

One of the requirements of designing such a system would be to estimate the size of the index, or database, that keeps track of all the backed up files and their locations. The size of this index is essentially 225 bytes times the number of times that each file is stored in the index. We first needed to decide on our backup schedule, since it determines how many copies of each file will be stored in the index. Our decision was to perform a monthly full backup, followed by a weekly level 1 backup and daily incremental backups. Also, by looking at our current backup logs, we estimated that 3-5% of all files would change within any given month. We also decided to keep three months of "browse" information online. At any given time, therefore, there would be approximately four to five references to each file stored in the index; we added one more for good measure. You then need to determine the number of files in your environment. We did this with the find command. This brought our

estimate to 13.5 GB². This will obviously need to be monitored over time for growth and performance.

Summary

The current design accomplishes the goal, even though it is not as fast as we originally hoped for. The only area for concern is that if any single system grows well beyond a terabyte, we will have to give up the "All saveset," and spread its full backups over two nights – something that we really don't want to do. It may then be necessary, to make the system go faster than it currently is capable of. If that becomes necessary, two possible enhancements could be explored.

The first would be to "go with the flow." A lot of work has been done in the last few years to make local backups faster with a much smaller CPU load than if they are performed over the network. This design ignores all those advancements because we are doing all our backups over the network. We did this for a reason, but perhaps putting backup devices on the large clients is the only feasible method once they approach a certain size.

The second possible enhancement would be to evaluate the Alteon (<http://www.alteon.com>) switches and NICs that support the "jumbo frames" of 9000 bytes. Alteon is claiming that they support a much faster transfer rate over Gigabit Ethernet with a much lower CPU usage. If that's true, the system could become over 100% faster just by changing which network we use.

Acknowledgments

I would like to acknowledge several people for their contributions to this project. Jeff Rochlin: for his input into the original design; The system would have looked much different without it! Rob Cotter: for his input into the final design and the chance to prove and implement the system. Celynn, Nina & Marissa Preston: for their understanding while I travelled to make this happen.

Author Information

W. Curtis Preston has six years of experience in systems management and is currently a Principal Consultant at Collective Technologies. He specializes in designing and implementing leading edge enterprise backup and recovery systems for both small and large companies. Curtis speaks about this topic around the country, and this is his second talk at LISA. He is also published in SysAdmin magazine, and is currently completing a book for O'Reilly and Associates. The book's working title is "Essential Backup and Recovery." Updates on the book (and other backup resources) can found at <http://www.backupcentral.com>. Reach him at curtis@colltech.com.

²10,000,000 files × 6 copies × 225 bytes = 13.5 GB

Configuring Database Systems

Christopher R. Page – Millennium Pharmaceuticals

ABSTRACT

This paper provides the system administrator with a fundamental understanding of database architecture internals so that he can better configure relational database systems. The topics of discussion include buffer management, access methods, and lock management. To both illustrate concepts in practice, and to contrast the two architectures of market leaders, Oracle and Sybase implementations are referenced throughout the paper. The paper describes different backup strategies and when each strategy is appropriate. In conclusion, the paper describes special hardware considerations for high availability and performance of database systems.

Introduction

Database systems are often at the core of enterprise computing infrastructures, and hence are almost always highly visible. A System Administrator (SA) unfamiliar with these complexities may find himself relying upon the insight of a database administrator (DBA) who may or may not understand a system's perspective. Becoming more common, the SA may even find himself playing the role of a DBA, a role that he may know little about. It is in the best interest of a SA that he has a basic understanding of database systems and understand some of the more important tradeoffs when configuring such a system. This paper provides some grounding for a SA who needs to support client/server relational database management systems (RDBMS).

Many of the concepts presented in this paper apply to nearly any relational database system and any flavor of UNIX. Where specifics are used, Solaris 2.5.1 is the operating system (OS) and both Oracle8 Database Server and Sybase Adaptive Server 11.03 are the RDBMS. Oracle and Sybase were chosen because they are the market leaders and are built upon different architectures. Most other RDBMS implementations are similar to one or both of these systems.

This paper is about database systems from the perspective of a SA/DBA dealing with the systems issues. The purpose is to explain to the reader why one would want to implement a system with a certain feature set, rather than explaining exactly what commands should be executed. This paper is not about database design.

Client/Server RDBMS

An RDBMS is charged with three tasks. One must be able to put data in, keep that data, and take the data out and work with it. A RDBMS manages resources much like an operating system. Most RDBMS perform their own memory management, can manage their own disks, and some even implement their own scheduler. Running a RDBMS is like running an OS on top of an OS. Though most RDBMS

implementations forego UNIX services in favor of their own, many of the concepts are shared.

In a client/server implementation of an RDBMS, the processing is split between a server computer and one or more client computers. The client computers concern themselves with presentation while the server is dedicated and can be tuned for raw computation. Communication between the client and server occurs over a network.

A simple view of RDBMS is that all operations are performed on and result in tables. A table contains rows of data, and each row may or may not have a value for each column that the table defines. While the data appears in tables, the RDBMS may store it differently.

There are two different classes of operations that one can perform on a database: on-line transaction processing (OLTP) and on-line analytical processing (OLAP). Decision support (DSS) or data warehousing are forms of OLAP. OLTP consists of many short transactions. OLAP consists of larger, longer running transactions. It is important to understand the workload when tuning your database. Performance goals will be either focussed on response time or throughput. A performance improvement in one of these areas generally negatively affects the other.

User's Perspective

The view presented to the user is important to understand because that is how the workload will appear to the system. Users access the RDBMS using data manipulation commands. They group the commands together into transactions to guarantee that the data is transformed from one valid state to another. Users operate within a schema, which is a set of tables they may access.

Relational Operations

The relational operations provide functionality to restrict, project, and join tables. The result of each of these operations is another table, and hence, the result of one operation can be used as input to the next. Restriction specifies on a per row basis which rows of

a table should be in the result table. Projection specifies which columns of a table should be returned. Join specifies how to pair rows of one table with rows of another to form the resultant table.

Structured Query Language (SQL) and Vendor Extensions

The Structured Query Language (SQL) is the standard language implemented by most RDBMS to access data. SQL is a declarative language, which means that the desired data is specified rather than an algorithm for calculating the data. The SELECT statement implements the relational operations, the INSERT command adds rows to a table, the DELETE command removes rows from a table, and the UPDATE command modifies values of columns in rows of a table. These four statements comprise the Data Manipulation Language (DML) of SQL. The Data Definition Language (DDL) defines database objects such as tables, view, and procedures. The Data Control Language (DCL) specifies user access to database objects and includes commands such as GRANT and REVOKE.

There are a number of concepts that permit the user to more easily work with relational data. Most SQL commands operate on a set of data, conceptually working with all rows of the table at once. Sometimes it is easier to operate on a row by row basis. The concept of CURSORS permits this. Two other useful concepts are VIEWS and STORED PROCEDURES. A VIEW contains no relational data itself, it creates a virtual table using a named sequence of relational operations on other tables and views. A VIEW can be used as shorthand for a complex and often repeated set of operations, a mechanism for security, and as a way to hide implementation. DML applies to VIEWS as well as to base tables. A STORED PROCEDURE is often used like a VIEW, but while a VIEW uses relational logic, a STORED PROCEDURE is procedural in nature. It can contain variables, operate on CURSORS, and contain conditional logic along as well as complex error handling.

Though SQL is a standardized language, most vendors provide a version of SQL that incorporates their extensions. Sybase's version of SQL is Transact-SQL (T-SQL). Oracle's version is Procedural Language/SQL (PL/SQL).

Transactions and Data Consistency

A transaction is a unit of work. It is an all or nothing operation, partial results cannot be seen outside of the transaction, and each transaction is independent of all other transactions. The changes made by a completed transaction are not available to other users until the transaction is committed. Before committing the transaction, the user may rollback the transaction and the state of the database will be as if the transaction never began. In an RDBMS, multiple DML statements can be grouped and treated as a transaction.

Each transaction is assumed to take the data from one valid state to another. Hence, if each and every transaction has been executed in full, or not at all, the database is considered to contain consistent data. Transactions can be aborted because a user decides the change would be invalid, or because the database is unable to guarantee that the transaction would operate independent of other transactions.

In Oracle, all DML executed in a session is treated as within a transaction until the transaction is committed or rolled back. In Oracle, a transaction will be considered committed when any DCL command is executed or it is explicitly committed using the COMMIT WORK statement. Sybase takes a different approach and treats each statement of DML as its own separate implicitly committed transaction unless the DML statements are explicitly grouped using the BEGIN/COMMIT TRANSACTION statements.

Database Objects

A schema is a set of database objects (tables, views, stored procedures, etc.) in a database. A schema can include multiple database objects, and a database can contain multiple schemas. Both Sybase and Oracle relate schemas to database users. The objects owned by a user in a particular database comprise that user's schema within the database. In practice, schemas tend to be more widely used in Oracle than in Sybase. This is because a Sybase server can support multiple databases. Whereas an application using Sybase will likely have his own database and create objects as the database owner in that database, an Oracle application will likely work within a schema in the instance's database.

Every database server defines a system catalog. This collection of tables, views, and procedures contains all the necessary information to access information in the database. Both the user, but particularly the DBA will often work with the system catalog in order to determine the status of the database. Oracle stores its catalog in the SYS schema. Some of Sybase catalog is present in each in every database on the server, but the server-wide catalog information is found in the master database. Sybase stores its system catalog in the database owner (dbo) schema.

The Server Perspective

A client/server database multiplexes users across the resources it manages and by acting upon the SQL instructions received from each of these users. It does this by converting the declarative SQL into execution plans. The execution plans are arrived at through the server's optimizer which takes into account among other things the number of tables being accessed, the potential sources (tables/indexes) of the data, the current storage structures in place, and the characteristics of the data being queried. The server, coordinated with other queries by the lock manager, then carries out the execution. It is through interactions with the lock

manager and the log manager that the user is provided a consistent view of the data.

Architecture

Client-Server Database systems fall into two different architectures: multithreaded and process pools (2-N). In both cases, a process resides on each client machine. In a multi-threaded architecture, each of these client processes speaks directly with the RDBMS server processes. In a process pool implementation, each client process communicates with a server process (a "shadow process") which then interacts with the RDBMS processes on behalf of the client. It is the presence of this "shadow" process that the architecture is sometimes called 2-N. All major RDBMS vendors provide or are working towards a multi-threaded implementation. An advantage of a multi-threaded implementation is that fewer OS context switches are required since the context switch occurs in the user space of one of the server processes which is servicing multiple clients. An advantage of the 2-N approach is that it more fully exploits UNIX's scheduling and has a less complex implementation.

Both Sybase and Oracle provide implementations that are hybrids. Sybase defaults to a single multi-threaded process but can be configured to run multiple instances of these processes, each communicating and coordinating with the other. Sybase calls each of these processes an *engine*. Oracle leans the other way, defaulting to a 2-N implementation. If configured as a *Multi-Threaded Server*, Oracle creates multiple shared server processes that act on behalf of multiple clients, as opposed to a one to one correspondence.

The structures used internally differ from one implementation to the next, but the purpose of these structures is similar. The structures in memory store buffers, execution plans, and state information for clients and server tasks. Structures on disk store the rows of data, meta-data, information for allocation accounting, and access structures such as indexes.

Oracle defines a server, also known as a database instance, as the coupling of its shared memory structures, defined as the *System Global Area (SGA)*, with four background processes (see Figure 1). The four

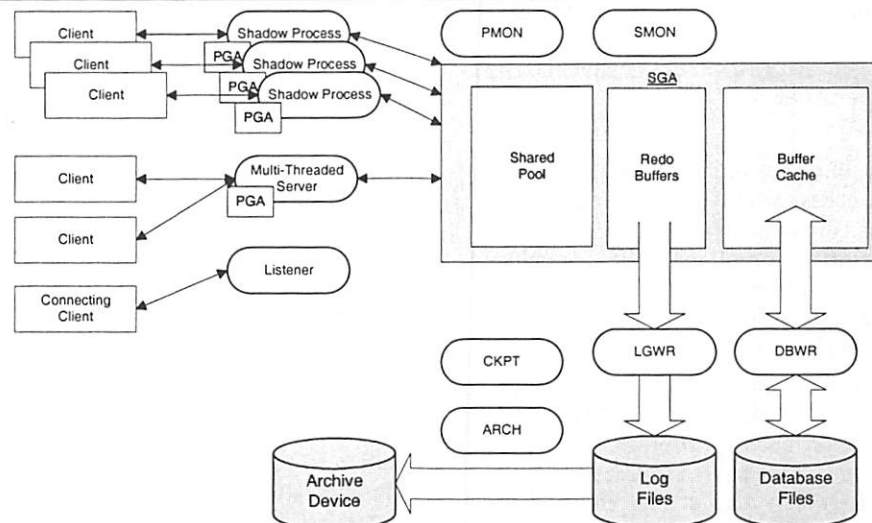


Figure 1: Four background processes.

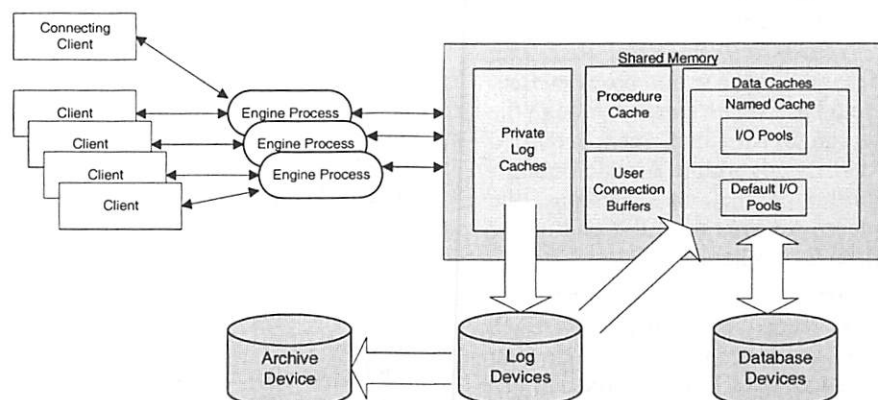


Figure 2: Process and database files.

processes are SMON, PMON, DBWR, and LGWR. SMON and PMON are for server and user process management respectively. DBWR is responsible for data file access. LGWR handles log file writing and checkpointing. Each server can access only a single database at a time. More functionality and performance is obtained by running additional processes such as CKPT to off load checkpoint processing and ARCH for log file archival.

A Sybase server minimally consists of one process and its database files (see Figure 2). A Sybase server can be configured for multiple engines, and can access multiple databases concurrently.

A Database in Operation

Once a session is created after the client has logged into the server, the database server receives the SQL from the client and parses it to ensure it is valid. The optimizer evaluates the resulting *parse tree* and consults with the available access structures and statistics to arrive at an *execution plan*. This execution engine follows the plan to obtain results and communicates them back to the client.

During the course of execution, the server performs buffer management, uses different access methods to locate data, and performs lock management to ensure transaction consistency.

Buffer Management

Databases would be performance limited by their I/O needs if a disk access were needed for each row access so caching is employed to reduce the number of disk accesses. The majority of the shared memory used by RDBMS is for buffering the data pages of the database and in general, more memory dedicated to buffering increases performance.

Databases implement caching algorithms to minimize the number of disk accesses. An MRU/LRU algorithm is a common caching strategy. Retrieved blocks of data from disk are chained together, with the most recently used (MRU) data at one end and the least recently used (LRU) data at the other. As the data is accessed in memory, the data block is moved to the MRU end. One problem with the MRU/LRU algorithm is that if there are unrelated applications accessing the database, or if massive tablescans are taking place, it is possible that the cache would prove ineffective because data would propagate quickly through the chain. Both Oracle and Sybase implement modified versions of the MRU/LRU algorithm. A major modification to the algorithm is that on tablescans, the incoming data is placed towards the LRU end of the chain so as to not flush the cache.

Because there are multiple processes that may need to access the data cached in memory, an RDBMS makes extensive use of shared memory. The amount of shared memory used by the RDBMS is fixed at initialization time. The server processes must be restarted to change the amount of shared memory used by the

server. The administrator may need to configure the amount of shared memory that is permitted on the system.

The DBA configures the size of the data block buffer region in Oracle by changing a parameter in the instance's *init.ora* file. A *Cache Hint* on a table in a query indicates that data from that table, even on tablescans, should be placed on the MRU end of the chain. The DBA can do this on a query by query basis or indicate this behavior for all operations.

The DBA configures the size of the shared memory in Sybase by using the *sp_configure* stored procedure. A Sybase feature allows this shared memory to be divided into a number of separate data caches. The DBA can bind unrelated tables to separate data caches to minimize the affects of one application on another. This is not only an effective way of isolating applications from one another but can also be used to make small tables memory resident. Tables that do not have bindings use the "default data cache."

Another important cache is where the SQL and execution plans are stored. Caching execution plans and code not only reduces the amount of time spent converting SQL into execution plans, but allows these plans to be reused by different sessions. Oracle calls this area in memory the Shared Pool and its size is explicitly set in the *init.ora* file. It is termed the *Procedure Cache* by Sybase and its size is defined as a percentage of memory allocated to the server.

Just as a table can be made memory resident in Sybase, a procedure can be locked (pinned) in memory using a stored procedure provided in one of Oracle PACKAGES, which is a grouping of stored procedures. The DBMS_SHARED_POOL.KEEP procedure can be used to pin the object in memory. Once pinned, the SQL remains cached in the library cache.

Before any change to data is made, enough information to recreate the change is written to a log file. This operation is called *logging*. Logging guarantees that even if the machine crashes while making the changes to the database, the server will be able to recover the changes. Sybase stores the changes in a *transaction log*, and Oracle stores the changes in the *redo files*.

In order to avoid scanning the entire log when doing a recovery, databases occasionally write checkpoints that summarize the status of the database. This *checkpoint* operation provides a shortcut to recovery. At checkpoint time, the database knows that all dirty pages have been written to disk. At time of recovery, the log is used to bring the database back to a consistent state. The system locates the time of last checkpoint and finds that position in the log file. It then rolls forward all completed transactions that occurred after the last checkpoint, and rolls back all transactions that were not committed, but began before the last checkpoint. Infrequent checkpoints lead to longer recovery time but less interference while running.

The log file is a potential point of contention within a database system because it is written to on every transaction. The log files should be on fast, dedicated disks. The log disk should not be used to store other database data because loss of the device would make complete recovery nearly impossible. Both Oracle and Sybase implement buffering on the log to increase performance. Sybase provides a *Private Log Cache* (PLC) that reduces the log bottleneck by providing each executing transaction with its own buffer rather than a shared buffer. Sybase further distributes logging by implementing a log for each of its databases, as opposed to a single log in the case of an Oracle instance.

A DBA has control over how often a checkpoint operation is done. At checkpoint time, all dirty data pages are written to disk and the log and data files are updated to reflect this fact. The potential performance gain achieved through infrequent checkpointing will be negated if the disk technology cannot handle the IO bursts. Infrequent checkpoints lead to longer recovery times, but are fine for data which is infrequently modified since only modifications are logged.

Sorting is a common operation in database systems either for application reasons, or for efficiency reasons. For example, an application may need to display data ordered by date. Another example is that in the absence of indexes, a sort may be needed in order to do efficient joins. Sorts are also needed when building indexes. Both Oracle and Sybase allocate special areas of memory for sorting and the amount of space is configurable.

Reading a block at a time from a database file can sometime be inefficient. It is advantageous if the server can be tuned to read multiple data blocks from disk for tablespaces. Both implementations support large I/Os for this purpose. Sybase can be configured to have multiple IO pools on a per cache basis, ranging from 2K (the default) up to 16K in power of 2 increments. Oracle can be configured to read multiple blocks by setting the `DB_FILE_MULTIBLOCK_READ_COUNT` in the `init.ora`.

Access Methods – Indexes

Indexes are used to quickly find data in a table, much as directories are used to find files in a file system. Databases can build multiple indexes over a table, which allows them to perform multiple types of lookups, for example being able to search by name or by date. However, extra indexes on a table can result in extra time being spent updating the indexes when the table is updated. Therefore, indexes should only be added when the query workload is going to be able to take advantage of them. In addition, different types of indexes are able to speed up different types of queries, so the appropriate type of index needs to be chosen for the workload.

When information is requested from a table in the database, the database system can scan all rows of

the table identifying and returning those rows that meet the criteria of the query. This operation, called a table scan, is sometimes the most efficient mechanism of accessing the data. It is efficient when the magnitude of the result set approaches the size of the table. For many operations, the result set is a small subset of the table. A table scan is very inefficient at these types of queries. An index can be used to make these queries more efficient.

An index is a structure in the database used by the system to effectively access rows of data meeting certain criteria. There are multiple types of indexes implemented by both Oracle and Sybase, and each type is optimized for different operations. In general, each instance of an index is specific to a single table but a table can have multiple indexes. An Oracle cluster is the exception to this rule, as it contains information from multiple tables and the underlying tables cannot have additional indexes.

An index stores an ordered replica of a subset of the information stored in the rows of a table as well as a pointer to the source row. In other words, it contains copies of certain columns from the row as well as a pointer to the row. The table is said to have an index on those columns. A query requesting rows from a table that has an index which meets the restriction criteria of the query will potentially use the index to determine specifically which data pages contain the requested information. Only those data pages need be referenced. A *covered index* is an index that contains all the columns needed by the query. When a covered query is used, the resultant data pages are not referenced. Since index pages contain a replica subset of information, typically more index rows can fit on a page than data rows can fit on a page. Effective index design can result in index scans rather than table scans, resulting in better performance. It is often an advantage to store indexes on separate devices to take advantage of these facts.

Why not create multiple indexes on every table? Sometimes one should, but the trade-off is space and wasted machine cycles on index maintenance. For each insert, the corresponding index entries need to be made and on every update, the effected indexes need to be updated. Index pages compete with data pages for cache space on the machine. One must be careful to weigh the tradeoffs. It is very important for the index designer to understand the goals of the system as well as the access methods. For smaller tables, indexes may never even be used! For example, if the entire table's rows fit on a single data page, that is a single I/O for each access of the table – an index will never be used.

Table and Index Structures

By default, a table is stored in a *heap* structure, which is a chain of data pages, with no particular order, and inserts are appended to the end of the chain. For more efficient access to the table data, some order

can be imposed on the table structure itself. Both Oracle and Sybase support the table data being stored in a B-tree format. In this format, the values stored in particular columns of the table determine the order in which the rows are physically stored. This type of structure is called a *Clustered Index* by Sybase and an *Index-Organized Table* by Oracle. Oracle also supports a *Data Cluster* structure in which the related rows of two tables are stored in the same data block.

Additional indexes can be constructed to improve performance. Since these are separate structures from the table, one should place these structures on separate devices than the source tables when possible to allow concurrent access. A *B-Tree* index is an ordered tree of index nodes that contains index entries pointing to the source rows in the table. The nodes contain the values of the columns that are indexed as well as the pointer back to the source row in the table. A *Bitmap* index stores the data values as bitmaps which allows quick access but is most valuable for indexes on columns contain only a small number of discrete values.

Statistics

Statistics are used by a database to decide how to access a table. For example, the optimizer uses statistics to decide which index to use, or even if an index should be used at all. Statistics are also used to decide the order for joining tables together. Therefore it is very important that the statistics be reasonably accurate. However, calculation of statistics is expensive, and they can be partially inaccurate without dramatically reducing performance.

Statistics are often overlooked and misunderstood by those new to database systems. Often, a DBA will observe that performance is non-optimal on a query because a table scan is being performed. The DBA then creates an index and observes that a table scan is still being performed. Sometimes, the optimizer chooses to perform a table scan over using an index because it believes the number of I/Os needed for the query will be fewer by doing the table scan. The optimizer would possibly be incorrect when basing its decision on outdated statistics of the table. The table's statistics includes how many rows are stored as well as breakdowns by value for each column used in the indexes. In the absence of statistics, default values are used. If the values are incorrect, it is very easy to understand why the wrong path may be chosen.

The system generates statistics by examining and classifying the data contained in the table. In Oracle and Sybase, the generation and refreshing of statistics is a manual operation. This is an often-overlooked task by new DBAs. For very dynamic tables, statistics need to be computed often, and static tables, not so often. Computing statistics is an expensive operation that can be a source of contention, as it needs to read the whole table. It is for these reasons that the task is not automatic. Oracle offers an optimized statistics

computation via statistic estimation. When used, Oracle reads every Nth row, and generates statistics based upon only these rows. This results in speedier statistics generation, but at the cost of potentially less than perfect statistics. Statistics do not need to be regenerated on read-only tables.

As an optimization, Sybase supports as an option, a one-time evaluation of execution plan on its stored procedures. In these situations, the execution plan is determined at the time of its first execution based upon the tables current statistics and indexes. This can result in increased performance as the generation of execution plan can be skipped on future executions. As data changes and new indexes are created, this execution plan may not be valid. Sybase provides a stored procedure `sp_recompile` that takes, as an argument, a table name. When statistics are computed on a table in Sybase, this stored procedure should be executed on the table so that all effected stored procedures will be reevaluated based upon the new information.

Lock Management

Lock management ensures that one transaction does not see the effects of another until the first transaction is committed. A lock on data limits operations on that data by other sessions.

There are different types of locks, and each permits and restricts operations. A *shared lock* permits other sessions to access the data, but prevents modifications to that data. An *exclusive lock* permits modification but prevents other sessions from accessing the data.

Lock contention can be a performance issue. Lock granularity is a way of reducing lock contention. The concept is that if one only needs to modify a single row, then the entire table should not need to be locked. Oracle supports row and table level locks. Sybase supports page and table level locking. Lock contention may result in a *deadlock* situation in which one sessions has locked data, waiting on another session's locked data, but the other session is waiting on the first's data. No processing occurs when this condition appears. The servers detect deadlocks and will terminate one of the deadlocked queries.

When an application is modifying a large number of rows of a table, the Sybase database may determine that it is more efficient to lock the entire table rather than contending with other applications and trying to obtain individual locks. When the optimizer determines that this is the case, and a table lock is obtained, it is called *lock escalation*. The DBA may tune a Sybase database by adjusting the thresholds at which escalation occurs. When an exclusive lock is obtained by a session on data, readers of that data are blocked until the lock is released. This is an efficient mode of operation when there are no other readers of the data, but inefficient when many readers are waiting on the lock.

Oracle attempts to improve performance by implementing *rollback segments*. Rollback segments store a version of data previous to modification so that it is available to other sessions. Performance is possibly improved because the exclusive lock obtained to modify the data does not block readers. This is an inefficient operation when there are no other readers of the data.

Data Files

Administrators need to specify for the database where it should store tables, indexes, logs, etc. They are stored in what Oracle calls data files and Sybase calls devices. The underlying storage system can be either file system, or raw devices. There are performance and convenience tradeoffs associated with these two choices as well as the choices of what type of hardware to use for the underlying storage.

Data that is often written should be on quick disks, preferably striped to utilize multiple spindles and write accelerated via battery backed up caching. Data that is read intensive can be accelerated via striped sets.

File Types

The raw data is the information that the user is interested in. It is in a format that the RDBMS understands, and includes space allocation accounting. It is beneficial if the tables being written to can be separated from those that are not, and the associated files placed on the appropriate devices. The use of volume managers may allow an administrator to move a logical device from one set of physical disks to another without affecting the IO path used by the database to access that logical device. If this is the case, the DBA can monitor the data file utilization of tables that are undergoing high levels of inserts and migrate from one device to another as the access patterns change.

Index files contain the structures that accelerate data access. Like plain data files, these structures are often read but undergo updates with each data update. Unlike plain data files, the DBA will likely be less sure about where in the index file the updates will occur because the files are structured and data distributed. If the table is read only, then the index devices need to be optimized for read access, otherwise, all files of the index must be prepared to handle the volume of updates.

Log files are where the transactions are stored. As the server must guarantee that writes have been written to log when each and every transaction commits, devices with log files on them need to deliver the highest performance on writes that is available.

Temporary space is needed for join operations, sorts, etc. Temporary space is to a database server as swap space is to a UNIX server and if there is a lot of activity on these devices, they should be optimized. Because this is temporary space, there may be an advantage to binding this space to system memory.

This must be weighed versus the performance gain that would be gained by dedicating the memory directly to the server where this memory might be better used for a larger sort area, for instance, which might decrease the requirement for temporary space. There are limits to the amount of memory that can be allocated as shared memory, so binding temporary space to memory is recommended when this is the case.

Unit of Storage

Files are divided into blocks by the database, and all database I/O accesses are at this level of granularity. The unit of storage in the database is a *page* or a *block*. A page size is 2K for Sybase and is user-defined for Oracle. Oracle's block size is 2K by default, but recommendations are at least 4K for OLTP applications and 8K or greater for OLAP applications. Once a size is selected, it cannot be changed for the database short of an export/import operation. The object-oriented aspects of Oracle are better suited for larger block sizes because the rows are typically larger with embedded arrays.

By default, data from disks are accessed one page at a time. Multiple rows of data can potentially be stored on a data page. Sybase imposes a limit on the size of a row so that it will fit on a single data page. The actual size limit is 2K minus some administrative overhead. Oracle permits a row to be chained across multiple data pages. This has the obvious advantage of not restricting row size, but has a downside in that multiple data pages need to be accessed in order to retrieve a single row. In Oracle, it is important to be on the lookout for excessive chaining as it can result in performance degradation. Certain conditions can result in a row chaining across multiple data pages, and sometimes this happens needlessly.

File System Versus Raw Devices

The database data files can reside on raw devices or on file system. Raw devices are more efficient, but file systems are easier to understand. One reason why one would choose to use a file system is when integrity is less of an issue (such as a development server) or the file system is optimized for database use and has addressed file system shortcomings such as the VERITAS Oracle Edition product. The discussions of file systems in this paper refer to conventional file systems unless otherwise noted.

File system devices offer the convenience of being more tightly integrated with the underlying operating system and are therefore easier to manage. Most people are more familiar with how to backup a file system data file than a raw device. The downsides of using file systems include the associated processing overhead of interacting with another layer before getting to the data. In order to access a row of data, the file system's structure must be navigated, possibly incurring two or more levels of indirection before arriving at the request data page. That page, e.g., an

index page, might then reference another data page which has to be retrieved, encountering the same overhead as the first reference. Some of this overhead may be minimized because the OS might be buffering the file system structure and data pages in memory. Still, the tradeoff is direct access to the data pages versus navigating the file system allocation structure. One must also consider that the memory used to buffer the file system pages might be better utilized by dedicating it to the RDBMS itself so that it could buffer data pages and minimize physical I/O to database data pages. An additional benefit to using raw devices is that you know that when a write says it is done, it is. There is less opportunity for hidden OS buffering which can undermine the integrity of the database data. Sybase recommends using raw devices and confronts you with a number of warnings at install time if you choose to place data on file system.

Mapping Tables to Data Files

Each database can be comprised of multiple data files, which may reside on different disks. In order to facilitate performance, both Oracle and Sybase give the DBA some control over where data is stored.

Oracle's concept is a *tablespace*, which is a collection of data files. A data file may be defined in only one tablespace. A tablespace for an object may be specified at object creation time. All data associated with that object will reside in the data files associated with that tablespace. Another concept, that of a *partition*, permits an Oracle table's data to be spread across multiple tablespaces.

A *segment* in Sybase is a tagging mechanism that identifies where data should be stored. An object created on a segment will store its data only on devices that are tagged as being part of that segment. A device can participate in multiple segments. Segments are important at the time of allocation. When a device is removed from a segment, the data that has already been allocated to that device remains.

Both Oracle and Sybase have limitations on the number of data files/devices that may be associated with a database. Up to 255 devices may be configured for a server, but only 128 per database. This can have a serious impact on very large databases (VLDBs) and requires special planning. Oracle's limit is over one million files per database.

Backups

There are multiple ways to backup a database. The simplest forms use standard UNIX utilities to do the backup of files but require system downtime. The more complete forms support backup of data files while the system is on-line, but are more complex. With this added complexity, recovery is possible up to the point of failure, whereas the other forms only support recovery to the time of last full backup. A final way to backup data is via logical import/export using tools provided by the database vendor.

To do a *cold backup*, the server is shutdown, and the SA uses whatever tools he likes to backup the files. The SA needs to use **dd** or similar utility in order to back up data that resides on raw partitions. Backing up database data files while the server is running is a recipe for disaster, unless the backup is coordinated with the server.

Cold backups are a viable option when the server can be shutdown while the backup takes place. Since a cold backup is a level 0 backup, the length of time required to do the backup grows as the database grows.

Because consistency is the corner stone of RDBMS systems, both Oracle and Sybase provide help for the DBA to backup a running database system. Oracle supports a *warm backup* that allows backups of tablespaces while the instance is running. During a warm backup, the instance continues to run, but individual tablespaces are taken off-line, backed up, and then put back on-line. The server must still be shutdown to backup the system tablespaces. Standard UNIX backup utilities are used during both cold and warm backups.

Warm backups offer some of the convenience of cold backups because standard utilities can be used to backup the database. They offer an advantage in that downtime for anyone application can be minimized if applications are partitioned into separate tablespaces so an application is down only when its tablespaces are being backed up. The total backup time for the entire server may be longer than that for a cold backup because the backup processes will be competing for resources with the database server itself as it continues to operate for other applications.

A *hot backup* is performed while the databases are on-line and available for use. This type of backup is required for 24x7 operations. This type of backup is not supported by default on either Sybase or Oracle servers. On both implementations, standard UNIX utilities are needed to back up the binaries and other non-database data file files.

A *Backup Server* needs to be installed in order to be able to do hot backups on a Sybase server. There must be at least one Backup Server per physical server, but multiple Sybase servers resident on the same hardware can share the same Backup Server. Once installed, a dump device can be created which tells Sybase which UNIX device to use for the purpose of backups. The DBA can then issue **dump** commands to backup the database to that device to stripe to multiple devices. The Backup Server handles all the book-keeping and coordinates with the server to ensure a consistent image of data as of the time that the backup began. The data files are completely available to user processes during this time period.

By default, when a checkpoint is performed on a Sybase database, the transaction log is truncated so that space is not wasted. As time passes and more

work is done, the transaction log grows. It records the modifications to the database as well as checkpoints. There is a point in the log where all transactions logged previous to this point have been committed or rolled-back, and the corresponding changes have been written to disk and a checkpoint performed. Transactions previous to this point are not needed by the database for recovery purposes. The truncation removes all these unneeded transactions that are stored in the log. By default, these logged transactions that are being truncated are discarded. Alternatively, this information can be written to disk and saved for the purpose of recovering a database to a particular point in time. In order to support up to the point of failure recovery in Sybase, the database must be configured to dump these transaction logs to disk. Sybase further places a restriction that the log device cannot also contain data for that database. Dumping transaction logs is configured by first disabling the truncate on checkpoint option for the database, and then by setting an automatic dumping of transaction logs either through threshold procedures on the log device, a CRON job that periodically dumps the log or through a combination thereof. Should the transaction log fill, the server will not permit further operations on the database until more space is made available in log, either through dumping transactions, or by allocating more space for the purpose of logging.

An Oracle database must be configured in *archive log mode*. When in archive log mode, another process called ARCH is started. This process monitors the redo files (which contain the transaction information). When the ARCH process determines that a redo file switch has occurred, it copies the inactive redo file to the archive directory or directly to tape, depending upon its configuration. In the *init.ora* file, the DBA needs to explicitly indicate where the log files should be archived, and what naming convention should be used. It is wise to have a number of redo files which are switched to in turn, because the ARCH process will not surrender the redo file back to the database for reuse until it has completed the archive process. This could stall the instance and give the appearance that the server is hanging.

When in archive log mode, a tablespace can be placed into backup mode. When in this mode, standard OS backups commands may be used to backup the tablespace's data files. Oracle provides special commands to backup Oracle control files so standard utilities should not be used when control files are active. This type of backup is sometimes called a *fuzzy* backup because it is possible that the OS copy will capture a data block write in progress, possibly capturing the pre-write version of the first part of the data block, and the post-write version of the second part. When in this mode, complete data pages are written to the log and this log file is used to get the consistent version of the data block so log activity needs to be monitored.

In Oracle, it is advised that unrelated schema reside in separate tablespaces. One reason for this is that the DBA may be asked to restore a schema's data from a previous date. If this schema is in its own tablespace, then the DBA may restrict access to this tablespace and restore all the data files from a previous date without impacting other schema's data. If two schemas are sharing the same tablespace and this request is made, the DBA must first backup the current tablespace, restore to the previous, export only that schema, restore to the original backup, and then import the exported data.

Consistency Checks

A database backup that contains database errors is just better than useless. It is important that consistency checks be performed on databases to ensure the integrity of data at the time of the backup. The downside is that consistency checks take time to perform and may lock tables in the process so coordination is essential.

Because each system has its own implementation, each vendor supplies utilities to help check data integrity. *Consistency checks* will, among other things, follow page chains and verify check sums. There are different levels of checking, and each provides the DBA with a different level of comfort regarding the integrity of the data. The checks should detect errors caused by faulty hardware as well as errors that are the result of buggy software.

The Sybase utility **dbcc** provides functions to check the allocation, catalog consistency, and index to table consistency. A new procedure available in Sybase 11.5 called **checkstorage** offers a very large performance improvement and is essential for VLDBs.

Oracle can check the structure of tables from within the system but also provides a utility called **dbverify** that operates on the raw database file and does not communicate with the instance. This utility will perform check sum computations on all blocks and report utilization within the data file.

Tuning

It is through a good understanding of the database server's workload that one can tune a database system. A SA can tune three separate layers of a database system: the OS, the database server, and the query itself. If a particular query is performing poorly, the DBA should first work with the query, then the server configuration, and then finally, the OS. Throughput issues should be addressed in the opposite manner, starting with the OS and working up to the individual queries.

Benchmark

Tuning a database system's performance is not unlike tuning any other system's performance. Before changing any parameters, the SA should benchmark performance for later comparison. The SA should use standard OS utilities to gauge OS performance and

database specific tools for query and instance performance.

Most database vendors supply performance tools specific to their implementation. Sybase provides **set statistics**, **sp_sysmon**, and **show plan** so that DBA can measure performance. Oracle has numerous **v\$ views** and **explain plan** to help the DBA understand underlying issues. Database performance tuning is a complex undertaking and is not explored any further in this paper.

Changing the Database Configuration

The method to view and specify system configuration is handled differently by each RDBMS. Oracle's configuration is specified by editing a text file called **init.ora** (or a variant thereof because in practice there may be multiple **init.ora** files, one per instance, so the server identifier (SID) is, by convention, embedded in the file named such as **initSID.ora**). One configures a Sybase server by executing the stored procedure **sp_configure**, passing arguments indicating which parameter to change allow with the value that it should be changed to. It should be noted that **sp_configure** generates a text file indicating will these new values and is read by the server at start up, similarly, in Oracle, one can use **show parameter** to view the current settings of the instance.

Hardware Considerations for Database Servers

High Availability (HA)

The purpose of HA is to minimize system down time. There are multiple options one may choose when configuring a system for high-availability and each has its associated costs. One attempts to achieve HA through redundancy at either the software or the hardware level. The redundancy can occur at the application level, system level, or at the subsystem level.

Making Disks Highly-Available

The most common form of redundancy for disks is implemented by using some level of RAID technology. This is often one of the more cost-effective approaches as a disk is the most likely component of a system to fail.

Assuming one is to implement RAID technology, a SA must decide how to implement it. There are both hardware and software solutions to choose from. Software solutions are provided by Sun and by VERITAS. Hardware solutions may seem to be a great solution, especially if the RAID controller offers redundancy. This may be misleading because if the redundant controller has a single point of failure, and if it fails, it will bring down the entire disk subsystem. This may seem unlikely, and probably is, but it can happen, and the author bears witness. For greater availability, the more redundancy that is provided, the better. No greater level of redundancy can be provided by a single system than by having disks mirrored across two separate controllers to two separate disk subsystems.

Most levels of RAID offer some protection from disk failure but the most protective is also the most expensive. There are also performance considerations as different levels of RAID are suited to different types of operations. It might be that RAID is not needed. Sybase itself can mirror its devices. Oracle does not offer mirroring at the data file level, but does offer to make multiple copies of its control and redo files. This has the added advantage that it offers some protection against accidental deletion so this option may want to be used even if another form of mirroring is used.

Even if data files are not stored on file system, it is likely that there will be some large file systems on a database server for error logs and archiving. The database server executables and configuration files will certainly be on found on a file system. If the database server does go down, the SA wants to make certain that the file system will be available and in a consistent state when the server come back.

A seasoned SA is familiar with file system consistency checks (fsck). The result of a fsck operation is not always pleasant, as some data may be lost. It is not always a timely operation either as it can take a long time on a large partition. A journaled file system is a solution to these problems. Similar to a database, the journaled file system logs changes to the file system so that, even if not brought down nicely, it can quickly recover and makes itself available. Still, some data may be lost, but the file system will be in a consistent state.

Making the Application Highly-Available

These solutions require that the system hardware be mirrored, and possibly sharing disks. In all cases, they require special preparations at time of system configuration.

One option is that the application is made highly available through the software itself. An example is that one can configure an Oracle Parallel Server (OPS) on two machines sharing disks. The load can be shared by the two machines and on failure, the other machine continues processing, taking up the load of the other. The details of this product are beyond the scope of this document, but it is important that a SA know that such a product exists.

Yet another option is to use clustering software which is independent of the application software. In these cases, the mirrored hardware shares the disks, and starts up the application when it detects that the primary host has failed. Veritas "First Watch" is an example of a product that offers this functionality.

A potential problem with both of these options is that someone must envision all possible modes of failure. If there is a class of failure which is not detected by the HA software, fail-over does not occur and the application remains unavailable. Two other types of failure are failure of the HA software itself, or a false

detection of application failure in which the secondary believes the primary service has failed but has not. This can lead to what is referred to as a "split-brain" in which both primary and secondary servers are attempting to master the service.

When primary and secondary computers are sharing disks, the most damaging type of failure is a faulty IO board in which, somehow, data is corrupted in route from the server to the disk. In a mirrored disk environment, it is possible that this corruption occurs before the logical mirroring resulting in corruption in the mirrors. The data is lost.

Another option is to employ software that replicates the data. Both Oracle and Sybase support replication between two separate systems with separate disk subsystems. In both cases, these are warm-stand-bys, requiring some intervention for fail-over and neither can guarantee that some transactions were not lost if the primary host experienced catastrophic failure.

Hardware Fault Detection and Spare Parts for Minimal Downtime

Sometimes the simplest solutions are the best solutions. Many systems provide hardware fault detection. When detected, the operating system will panic, and upon reboot, these systems are able to disable the faulty hardware. This is a viable solution provided that your system can tolerate such downtime and can perform with limited resources. A pile of spare parts can help should your system be able to endure brief interruptions but needs all resources.

The Sun Enterprise server series of computers is an example of hardware that can detect and offline faulty CPUs and memory. After panic, the system comes up with the faulty part disabled. This type of functionality lends itself to environments that need to minimize downtime.

Performance

One of the basic tenets of performance tuning is that there are interdependencies between the variables. The performance triangle has corners labeled CPU, memory, and IO. This section briefly describes how these resources are important to a database system. A change in one of these aspects is likely to affect the performance of the others. Generally speaking, adding more CPUs requires more memory, adding more disks or network requires more CPU to service the interrupts. A fully utilized system will be very sensitive to these changes, a less utilized system will be less sensitive. In practice, one bottleneck is typically traded for another. These concepts hold true for a database system as well as they do for any other type of system.

CPU

A system is CPU bound when all of its CPUs are consistently busy doing user privileged processing. Standard OS utilities can be used to measure CPU utilization in Oracle as well as Sybase. Since Sybase is multi-threaded, more insight is needed. Sybase's

`sp_sysmon` provides CPU utilization from the server's perspective. When Sybase indicates that more power is needed, but the OS reports under utilization, the problem may simply be solved by configuring more engines. More CPUs should be added to an SMP system when it is CPU bound.

Adding more CPUs almost always helps a CPU bound system. When adding more CPUs, do not forget that this also increases the load on the back plane in an SMP environment and may result in the need for more memory. If the application is CPU intensive, and there are many CPUs, the system may be spending much of its time dealing with cache coherency. Sybase 11.5 allows you to specify a lazy LRU strategy that can help in this situation. The lazy LRU algorithm does not have to update the in memory buffers as frequently because pages are not shuffled throughout the page chain as they are accessed. This algorithm results in fewer cache invalidations during the system maintains hardware cache coherency.

Memory

Memory is probably the best way to increase the performance of a database system. Oracle's installation manual recommends that the system's memory be backed by three times as much swap space. It is the author's recommendation that if anything close to that amount of swapping is observed, then more memory is needed. A Sybase system on Solaris should not do any swapping as it has a fixed number of processes and uses intimate shared memory (ISM). ISM is locked down and cannot be swapped out. A requirement of ISM on some UNIX systems is that ISM must be backed by sufficient swap space even though it will not be swapped. If the swap space is not provided, the system may use simple shared memory and performance will suffer, with no other warning. This condition can be detected by observing the shared memory allocations when using a system call tracing utility on the database process.

With Sybase, as the number of user connections increases, so does the amount of memory required. This, like many other parameters, is a static parameter in Sybase and must be monitored and set at an appropriate level. The output of `sp_configure` indicates how much memory is being statically allocated.

Both Oracle and Sybase provide guidelines for modification to system configurations for shared memory, semaphores and other memory structures. Very little guidance is given relative to these values. What is one to do if more than one instance of the database server is going to be run on the hardware? The `ipcs` command can be used to determine if the IPC resources are being appropriately used. It can be used to print information about each of the constructs including when the construct was last accessed, and its current size. This output can then be compared to what has been configured and tuned down if the numbers indicate over configuration. Both Oracle and Sybase

will complain very loudly if the needed resources are not available. As with any tuning exercise, changes should be made during a maintenance period and measured against representative loads.

Input/Output (IO)

Both disk and network IO is important to the performance of a database system. Each disk and network IO requires that the CPU service it. The more data that can be transferred in a single operation, the better the performance. When tuning a database system, after adding more memory, IO is the place to start.

The disk technology used should be matched with the needs of the files that reside on it. Earlier discussions of file types indicate the needs of each. RAID technologies offer different combinations of performance, fault-tolerance, and value. In general, RAID 0 is an excellent and inexpensive solution when fault-tolerance is in issue. The striping provided by RAID0 put many disk heads to work on each operation. RAID1+0 may not always perform as well as RAID0, but provides the extra fault tolerance that will be needed in 7x24 operations. RAID5 is a good compromise on read-only systems.

Some disk controllers offer battery backed-up cache. This can be a huge performance win provided that it is not a write-through cache. It is likely that the cache would aid writes but would do so less frequently on reads as most database systems already provide complex caching algorithms. There is also some preliminary evidence that caching at the controller will improve latency but can in some cases lower throughput for certain transaction types. Another alternative is solid state disks for data that is accessed frequently.

Database client and server communicate over the network. Both Sybase and Oracle allow you to configure multiple network listeners so that the network load can be distributed over multiple interface cards.

The OS network parameters can affect the database system performance. Be sure to check out the keep-alive setting using **ndd**. An undetected lost network connection can keep data locked and hang up a system.

To better utilize the network, Sybase supports different packet sizes. The output of **sp_sysmon** indicates the average packet size sent and received. Using a size that takes this average size and the physical network packet size into account can offer some performance gains. Additional memory must be dedicated for network buffering if the size is changed. Since this memory is statically allocated, it must be monitored closely as it will be wasted if there are many unused connections. Performance will suffer as sessions are forced to use default sizes if not enough memory is allocated for this purpose.

Summary

With an understanding of the internal workings of a database system, a SA should be better able to effectively communicate with and possibly even cover for a DBA. With knowledge of the different file types, a SA can place the appropriate files on suitable disk drives. An understanding of the internal structures and processes will help the SA better allocate and monitor system resources. It is only through an understanding of database working and load that an SA can properly configure a database system.

Author Information

Christopher R. Page is a Principle Systems Engineer at Millennium Pharmaceuticals, Inc in Cambridge, MA where he implements and maintains systems which support the company's drug development efforts. Chris obtained his MSCS from Boston University and his BSEE from WPI. Before joining Millennium, Chris was a firmware engineer at Raytheon where he worked on Air Traffic Control and Microwave Landing Systems, and a software engineer at Lockheed Sanders where he worked on surveillance systems. He has over six years of professional experience with RDBMS. Reach him at page@mpi.com.

References

- [1] Adams, Kevin, *Release 8.05 for Sun SPARC Solaris 2.x Installation Guide*, 1998
- [2] Aronoff, Eyal, "Oracle Block Size: Larger Is Better," *IOUG-A Live! Conference Proceedings*, 1998
- [3] Aronoff, Eyal, and Loney, Kevin, and Sonawalla, Noorali, *Advanced Oracle Tuning and Administration*, 1998.
- [4] Bobrowski, Steve, *Oracle 8 Architecture*, 1998.
- [5] Cockcroft, Adrian, and Petit, Richard, *Sun Performance and Tuning, second edition*, 1998.
- [6] Corey, Michael J., and Abbey, Michael, and Dechichio, Daniel J., and Abramson, Ian, *Oracle8 Tuning*, 1998.
- [7] Date, C. J., and Darwen, Hugh, *A Guide to the SQL Standard, fourth edition*, 1997.
- [8] Gray, John Shapely, *Interprocess Communications in UNIX, second edition*, 1998.
- [9] Kirkwood, John, *Sybase SQL Server 11, an Administrator's Guide*, 1996.
- [10] Koch, George, and Loney, Kevin, *Oracle8 Complete Reference*, 1998.
- [11] Silberschitz, Abraham, and Galvin, Peter Baer, *Operating system Concepts, fifth edition*, 1998.
- [12] Mauro, Jim, "Inside Solaris: Shared Memory Uncovered," *SunWorld*, September, 1997.
- [13] Veritas, www.veritas.com, 1998.
- [14] Wong, Brian L., *Configuration and Capacity Planning for Solaris Servers*, 1997.

A Configuration Distribution System for Heterogeneous Networks

Glédson Elias da Silveira – Federal University of Rio Grande do Norte
Fabio Q. B. da Silva – Federal University of Pernambuco

ABSTRACT

This article presents a configuration distribution system that assists system administrators with the tasks of host and service installation, configuration and crash recovery on large and heterogeneous networks. The objective of this article is twofold. First, to introduce the system's modular architecture. Second, to describe the platform independent protocol designed to support fast and reliable configuration propagation.

Introduction

Service configuration is one of the most common tasks for the system administrator. It is also one of the most critical, since it impacts network performance, resilience, predictability, and security. This task becomes increasingly difficult as the size and heterogeneity of the networks increase. As pointed out in [1], some of the reasons for this increase in complexity are:

- configuration of each service must be uniform over the network, requiring update on every host anytime a configuration change is required;
- in general, there are great differences in the actual format and location of the configuration files of a service for each platform. Therefore, to configure a service in a heterogeneous network amounts to configure the service in each platform;
- each operating system provides its own set of non-standard configuration parameters for the common network services. In most cases, it is necessary to learn and use these non-standard features to achieve optimal performance of the service in each platform;
- usually, large networks are managed by a team of system administrators. Configuration rules and parameters must be effectively communicated among team members to avoid inconsistencies that can arise due to personal preferences during the configuration process.

Several works have proposed methods and tools to assist service configuration and management [1,2,3,4,5,6,7]. A common underlying characteristic of those works is the existence of a (central or distributed) repository of configurations. The possibilities of configuration consistency checking and easy recovery of host configuration after a serious system's crash are just a few of the advantages of having a configuration repository. In [1], the repository holds configuration for various services and allows consistency analysis to be performed among different services.

Once service configurations are stored in a repository or database, they must be propagated to target hosts on the network. This article addresses the problem of distributing service configuration from a (possibly distributed) database to network hosts. It presents a configuration distribution system and a propagation protocol for configuration distribution in large heterogeneous networks. Hereafter, the system and protocol are referred to as CDS.

The following section provides an overview of configuration management and establishes the scope of the work presented in this article. Later, related work is compared to the approach of this article, and the CDS modular architecture is presented. Then, the propagation protocol is described in depth and some details of the CDS implementation are provided. Finally, some conclusions are presented.

An Overview of Configuration Management

The Network File System (NFS) [8] is a widely used service that provides file sharing among hosts in the network. NFS is used here to illustrate the problem of configuration management. The following steps are usually performed to manually configure the NFS service:¹

1. Login on the server and edit the NFS configuration file to export the desired filesystems to client hosts.
2. Initialize the NFS daemons on the server.
3. Import desired filesystems on the client, either manually or by editing the configuration files.
4. Under demand, unmount filesystems on the client side.

This manual process only works for small networks or in situations in which changes in the configuration of servers and clients are seldom necessary. However, even in those situations system administrators prefer to use some automated support to avoid inconsistencies and insecurities that may arise from

¹The configuration of most services uses similar sequence of steps.

human error. Furthermore, this support can be critical during crash recovery, when a machine must be promptly reconfigured to its state prior to the crash. Ideally, to configure the NFS on large and critical systems, the administrator should perform a number of tasks in a coherent and consistent form:

- **Service planning:** to define the file systems exported by the servers and imported by the clients, as well as, their access and security characteristics. This task should abstract away from platform specific features.
- **Configuration consistency checking:** the planning of the service must be carefully checked for consistency. For instance, it should not be possible to plan a system in which clients import a filesystem that is not exported by any server.
- **Generation of configuration files:** from a consistent service plan, the NFS export and import files should be generated. At this stage, platform specific features must be used to create files on the right format for each supported operating system and hardware platform.
- **Configuration propagation:** the export and import files must then be distributed to the corresponding hosts.
- **Configuration activation:** the necessary actions must be performed on each host to activate the new configuration. It may be necessary to reboot the host.

The CDS supports configuration propagation and activation, and provides a platform to be used by any system implementing the task of configuration planning, consistency checking and generation of configuration files. A system that supports these tasks is fully described in [9]. This system uses the CDS as its configuration propagation and activation mechanism.

A Comparison with Related Approaches

Recently, several systems were developed to assist the task of host/service configuration and management [2,3,4,5,6,7,10]. Generally, these solutions were designed to resolve local requirements. However, their technologies identify and evaluate various characteristics of the configuration process. There are many ways to classify and evaluate configuration distribution systems. This section addresses and compares the main techniques used to distribute and activate service configuration from a database to network hosts.

The configuration propagation can be performed using several distribution mechanisms. Other works [2,3,4,5,6,7,10] have used the following approaches: remote commands, TCP/IP socket, NIS, NFS or a specific protocol developed using the Remote Procedure Call (RPC) mechanism.

In UNIX systems, the *rcp*, *rsh*, and *rdist* commands can be used to allow remote copy and

processing. The main drawback of *r*-commands is the need for privileged access on the remote hosts, which is a major security drawback. *Config* [4] uses *rdist* to distribute the configuration files from the repository to the network hosts. This repository is created in a server host as hierarchical directories, which can be replicated in other hosts. OMNICONF [6] also uses *r*-commands (in particular *rsh*) to manipulate files and directories that keep the configuration of the hosts in a central database.

Mechanisms based in the client/server model can be developed using the socket interface presented in TCP/IP environments. Its disadvantage is the requirement to manipulate the data formats of each platform. The main advantage is to require only the TCP/IP protocols running for using the solution. In addition, they do not demand privileged access on the remote hosts. However, it is very difficult and complex to work directly using the socket interface.

Other approaches have used a combination of NIS and NFS to distribute configuration information. The lack of an incremental update mechanism causes serious performance and network-traffic problems when using NIS in large networks. Moreover, NIS has notorious security problems. Furthermore, to use these distribution mechanisms requires all services on which they depend, to be up and running before any configuration distribution can be performed. Another drawback with NFS is to limit the database to be implemented using only flat files.

The *lcfg* [2] uses NIS to distribute configuration files that are stored in the central database. GeNUAdmin [3] propagates the configuration information using the *rsh* command, and retrieves data and programs using the NFS. Its database is centrally implemented with directories and files. According to the paper, it seems that *Gutinteg* [5] uses the NFS to propagate the configuration files and software packages, which are stored in directories and files in a central server.

Another solution is to design and implement a distribution protocol using RPC. One clear advantage of this approach is that the configuration distribution system only needs the RPC system to be running. Therefore, it can be used to configure every service that is started up after the RPC system in the boot process. Besides, a simple and clear design can lead to a safe and robust protocol because it uses a standard to represent external data and does not need privileged access on the remote hosts.

AUTOLOAD [10] and *Aurora* [7] use RPC based protocols. The former implements the mechanisms to propagate configuration and software package information from a central database that is implemented using flat files.

For the reasons cited above, the design and implementation of an RPC based protocol is the most interesting approach, despite its inherent complexity.

This is also the approach used to implement the system presented in this article.

Almost all the systems presented above make use of a central database structured as a hierarchical set of files and directories. The system that is proposed in this article does not impose any database organization or architecture. The database can be implemented using a relation/object-oriented DBMS or flat files. Moreover, it supports several architectures: central, distributed or replicated.

When a configuration is modified in any host, the systems cited above use either a pull or a push mechanism, never both. CDS supports both pull and push mechanisms to propagate configurations.

The CDS Architecture

CDS allows host and service configuration information to be consistently defined, stored in a database and propagated to the corresponding hosts on a heterogeneous network. The propagation of the configuration information is controlled by a uniform, simple and efficient protocol, designed especially for the system.

Using automated tools to define the service configuration, the network administrator initializes the configuration process of a given service. After this initialization, the protocol reads the configuration information from the database and transfers it to the hosts, where specified operations are performed. The database provides the following features to the system:

- **Consistency and uniformity of the configurations:** achieved through the verification of configuration information stored in the database, according to pre-defined rules also coded in the system.
- **Automatic host and service reconfiguration:** enforced by the configuration stored in the database that stays available even if the host is down (provided the database server is running).
- **Large-scale network scalability:** accomplished by the automatic propagation of the configuration information.
- **Easy inclusion of new services and platforms:** supported by the capability of defining meta-configurations in the database.

Figure 1 shows a schematic view of the CDS architecture and its components. Each component of the

architecture is discussed below:

- **Administration ToolBox (ToolBox):** a set of integrated tools used by the network administrator to define and configure hosts and services in the database.
- **Configuration Database (Database):** a data repository that holds information about the configurations and their distribution. For each configuration, the Database holds information about the target hosts and the actions that must be executed in each host after propagation. The administrator manages the Database using the automated tools presented by the ToolBox.
- **Configuration Protocol (Protocol):** component that performs the distribution over the network of the configuration information stored on the Database to the hosts. It has two sub-components:
 - **Configuration Protocol Server (Server):** software component responsible for reading the configuration information from the Database and propagating it to the corresponding host through the Protocol. The architecture supports several Servers in a given network to obtain high degree of performance and resilience.
 - **Configuration Protocol Client (Client):** software component that interacts with the Server to receive the configuration information stored in the Database and performs the operations to configure the host on which it is running. Every host on the network can be managed as long as it executes this component. In fact, hosts can run both the Server and Client components if necessary.
- **Database Access Interface (Interface):** component used by the Server to interact with the database management system (DBMS) to access the configuration information held in the Database.
- **Trap:** mechanism that allows the ToolBox to indicate the presence of modifications in the configuration of a given host. This element defines an immediate method of distributing configuration.

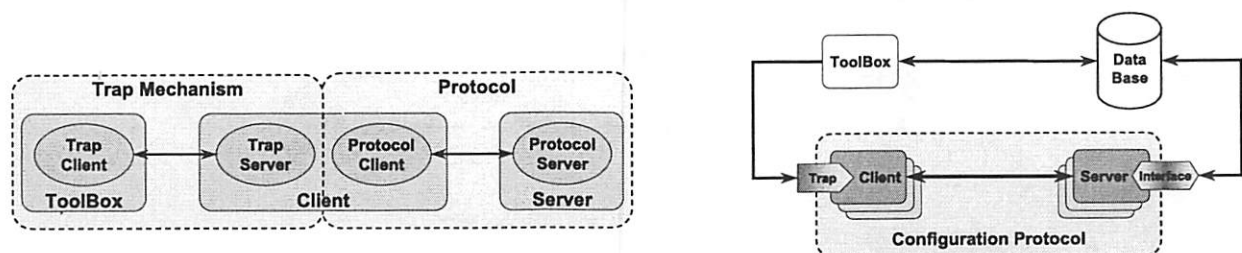


Figure 1: System's Architecture.

The system's architecture provides the independence between its components, allowing the design and implementation of each component to be performed in an autonomous way. This independence is supported in the following way:

- **Protocol/Database:** the operations performed by the Interface implement the independence between the Protocol and Database. To use the Protocol with a given DBMS, only the Interface must be modified, preserving the Client and Server codes.
- **Protocol/ToolBox:** the model and structure of the configuration information stored in the Database implement the independence between the Protocol and the ToolBox.
- **ToolBox/Database:** another interface can be used to implement the independence between the ToolBox and Database. This interface has not been implemented in this version of the system.

The FLASH Project [12], where this work is developed, has activities centered in each system's component. The following section presents the Protocol's architecture, components and operations.

The Configuration Protocol

As discussed before, the Protocol was designed using the Remote Procedure Call (RPC) paradigm. RPC allows access to remote services through a procedure-oriented interface, which is based on the client-server model. The RPC server is a program that implements a set of procedures that are called by the RPC clients. A program number and procedure numbers internally represents the program and its procedures.

To support differences in data representations among several platforms and network technologies, RPC uses the External Data Representation (XDR) standard, which provides common data representation over the network. RPC/XDR ensures portability across

different hardware platforms, operating systems, network architectures and transport protocols.

Activation Modes

The configuration propagation must be robust even in conditions of high network and CPU load. For this reason, the Protocol ought to reduce the traffic over the network and the processing load in the hosts. In addition, the Protocol must offer a mechanism for immediate configuration of services. The Protocol reduces the traffic load using messages that carry a small amount of information. Furthermore, the Protocol's operations generate negligible traffic when the host does not require modification in its configuration.

The network's resources are used more intensively only when an action requires the transference of a file stored in the Database. However, in this case, the Protocol divides the file in blocks that are transferred over independent messages. This message partitioning avoids peaks of network traffic.

To minimize CPU load on the Servers and to allow the immediate configuration of services, the Protocol has two activation modes:

- **Pull Mode:** the Client periodically activates the Protocol at time intervals defined by the administrator. In such case, the activation control is distributed among the Clients, and thus, reduces the processing load on the Servers.
- **Push Mode:** the Client activates the Protocol upon the receipt of a trap signal sent from the ToolBox. This signal forces the immediate propagation of a new configuration towards the corresponding Clients.

Structure of the Configuration Information

The Protocol employs the concepts of tasks and actions to define the configuration structure, as shown in Figure 2. A host configuration is an ordered set of tasks, where each task is responsible for configuring a given service. A task defines an ordered set of actions, where each action performs a phase in the configuration process of a given service, and thus, must be

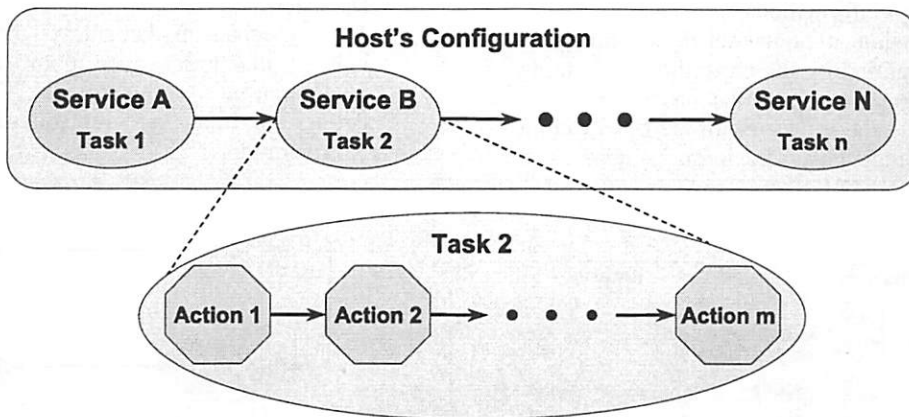


Figure 2: The Configuration Structure.

transferred to and sequentially executed on the Client side. Upon the execution of all tasks, the services of the Client host are properly configured.

For instance, to configure a given NFS server to export a new file system, the actions would be first to copy the modified NFS export file onto the server and then either to execute a command to export the new file system or reboot it and activate the changes.

The result of executing the actions is the installation, execution or removal of programs, scripts or configuration files on the Client. There are five types of actions implemented by the Protocol:

- **Execute:** executes a binary program on the Client.
- **Script:** interprets a shell script on the Client.
- **Copy:** transfers a file to the Client.
- **Remove:** removes a file from the Client.
- **Reboot:** reboots the Client.

The *Reboot* action can appear in any position of the action list of a given task. When a *Reboot* action appears in the middle of an action list, the Client is rebooted and then resumes the execution from the first action that follows the *Reboot* in the list. The *Execute*, *Script*, and *Copy* actions perform their operations over a file identified in the action. The Protocol defines that this file can be stored in the following places:

- **Client's File System:** the file is directly manipulated by the action.
- **Configuration Database:** the file must be transferred from the Database to the Client, and then, manipulated by the action.

Architecture of the Service

The Protocol is based on the RPC's client-server programming model, where the RPC client requests the configuration information and the RPC server processes the request and transmits the result to the client. Hence, as illustrated in Figure 3, the RPC client and server are implemented on the Client and Server, respectively.

The trap mechanism is also based on the RPC model. In such case, the RPC client sends the trap signal to the RPC server, which processes the signal initializing a new Protocol iteration. On that account, as shown in Figure 3, the RPC client and server are implemented on the ToolBox and Client, respectively.

The Protocol uses the concept of stateless Server, which does not need to keep information about the Protocol's state on the Clients. This feature allows the recovery from failure on the Server to be performed in a simple way.

The Protocol's Operations

The Protocol is defined by a set of procedures called by the Clients and processed by the Servers. These procedures are synchronous, that is, once the procedure finishes its execution, the Client can assume that the operation has been completed and any data bound to the request are stable in the Database.

The Protocol is defined by the following procedures:

- **Request Server:** identifies the Servers currently available on the network.
- **Send Program:** stores in the Database the RPC program number used by the Client to process the trap signals.
- **Request Tasks:** retrieves the task list to be configured on the Client.
- **Request Actions:** retrieves the action list of a given task.
- **Read File:** retrieves from the Database a file associated with an action.
- **Confirm Action:** stores in the Database the successful conclusion of an action.
- **Confirm Task:** stores in the Database the successful conclusion of a task.
- **Confirm Configuration:** stores in the Database the successful configuration of the Client.
- **Send Error:** stores in the Database some information about an error identified by the Client while an action was being performed.

The Protocol represents each host, task and action using a single identifier stored in the Database. These identifiers are used as arguments for and answers from the procedures. Their representation formats are dependent of the DBMS used to implement the Database. Therefore, the Clients and Servers must manipulate the identifiers as a set of non-interpreted bytes that indicate the host, task or action manipulated by the procedures. The ToolBox generates the identifiers when a host, task or action is created in the Database. These identifiers implement a simple protection mechanism since the Server can check, for

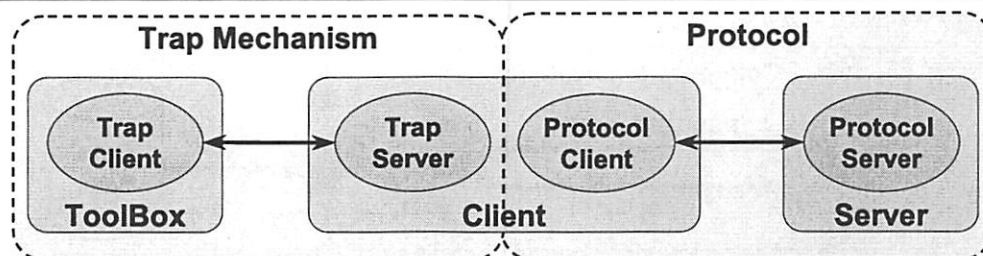


Figure 3: Service Architecture.

instance, if a given action belongs to the specified task or a given task belongs to the specified Client.

The Configuration Process

In both pull and push mode, the configuration process is implemented through the following sequence of operations on the Client side:

- **Select Server:** Client calls the *Request Server* procedure to identify Servers currently available on the network (Figure 4). This procedure must be called in broadcast mode, allowing every available Server to respond. Once the Servers have been identified, the Client selects one using an availability criterion. If no Server responds, the *Request Server* procedure will be called after waiting for a random period of time.
- **Send Trap Program Number:** Client calls the *Send Program* procedure to register in the Database the RPC program number used to process traps (Figure 5). This procedure is called only during the first activation of the Protocol.
- **Request Configuration:** Client calls the

Request Tasks procedure to request the task list $[T_1, \dots, T_n]$ to be configured. The Server retrieves the Client's task list from the Database and transfers it to the Client. If the Server does not find any task to be configured, it will send an empty list. In such case, the Client finishes the current interaction (Figure 6).²

- **Process Tasks:** for each task T_i , the Client sequentially requests, performs and confirms each action that composes its action list $[A_{i1}, \dots, A_{im}]$. This is explained in detail below:
 - **Request Action List:** Client calls the *Request Actions* procedure to request the action list $[A_{i1}, \dots, A_{im}]$ of the task T_i . The Server retrieves the action list of the task T_i from the Database and transfers it to the Client. If the Server does not find any action to be performed, it will send an empty list. In such case, the Client must confirm the task calling the *Confirm Task* procedure.

²All other procedures have similar interaction model..

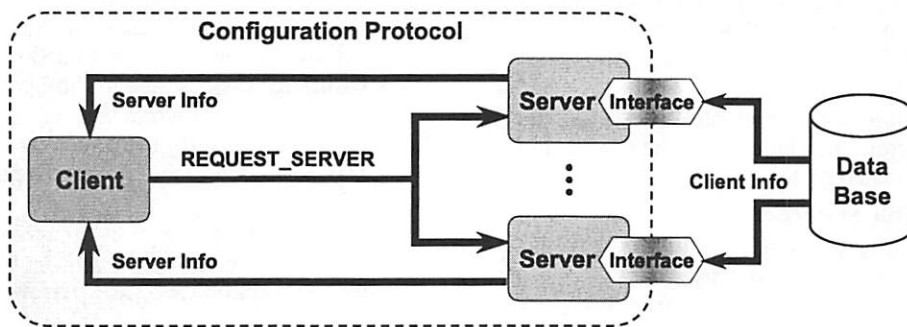


Figure 4: Select Server.

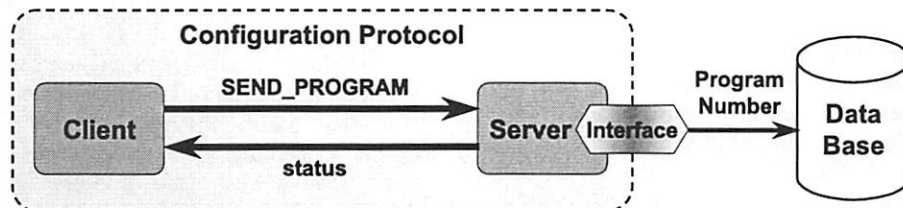


Figure 5: Send Program Number.

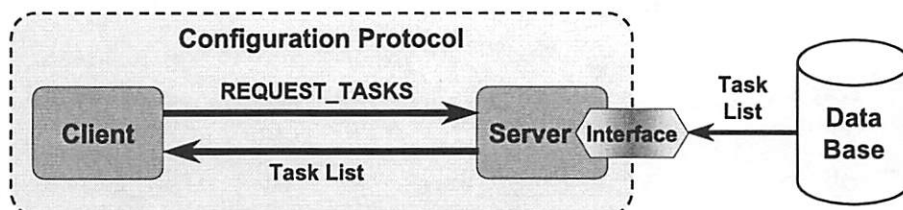


Figure 6: Request Configuration.

- **Process Actions:** Client sequentially performs and confirms each action A_{ij} of task T_i :
 - **Perform Action:** Client performs the action A_{ij} executing the specific functions of the action type. In *Execute*, *Script*, and *Copy* actions, if the file to be manipulated is stored in the Database, the Client will use the *Read File* procedure to read data blocks of the file.
If the Client detects an error during this phase, it will call the *Send Error* procedure to register in the Database an information about the error. Immediately after the error is registered, the Client suspends the current iteration and disables the pull mode of the Protocol's activation. By doing that, a new iteration will be started only upon the reception of a trap. This behavior avoids the Client to be kept trying to perform an action that has an error. The administrator must correct the action and then send a trap signal to the Client.
 - **Confirm Action:** upon the successful conclusion of the action A_{ij} , the Client calls the *Confirm Action* procedure to confirm the action in the Database.
 - **Confirm Task:** if all actions of the task T_i were confirmed, then the Client calls the *Confirm Task* procedure to confirm the task in the Database.
 - **Confirm Configuration:** if all tasks of the Client were confirmed, then the Client calls the

Confirm Configuration procedure to define the conclusion of the configuration in the Database.

In case of failure of a given Server, the Client can continue the current Protocol iteration with another Server, exactly from the point where the failure has happened. To do that, the Client must identify other Servers using the *Request Server* procedure, and then, continue the interaction with the selected Server. So, when the Client receives an RPC error, it must call the *Request Server* procedure to identify another Server and continue with the Protocol iteration.

The confirmation procedures allow a given Client to reboot and continue the configuration process without redoing actions and tasks previously confirmed. In such case, when the Client calls the *Request Tasks* and *Request Actions* procedures, the Server informs only the tasks and actions that were not configured yet.

The Trap Mechanism

The trap mechanism allows the administrator to indicate modifications in the configuration of a given host. It is composed by a single RPC procedure. This mechanism forces the Protocol activation without waiting for the pull mode. Taking into account the Client's state when a trap is received, its processing can be performed in two ways:

- **Immediate Mode:** the Protocol is immediately activated on the Client when the trap is received during the waiting time of the pull mode.
- **Delayed Mode:** the Protocol activation is delayed in the Client when the trap is received during an iteration of the Protocol. In this case, the trap must wait the conclusion of the current iteration, and then, it activates a new iteration.

The trap mechanism has a single RPC procedure, called *Send Trap*, which is responsible for sending the signal from the ToolBox to a given Client (Figure 7).

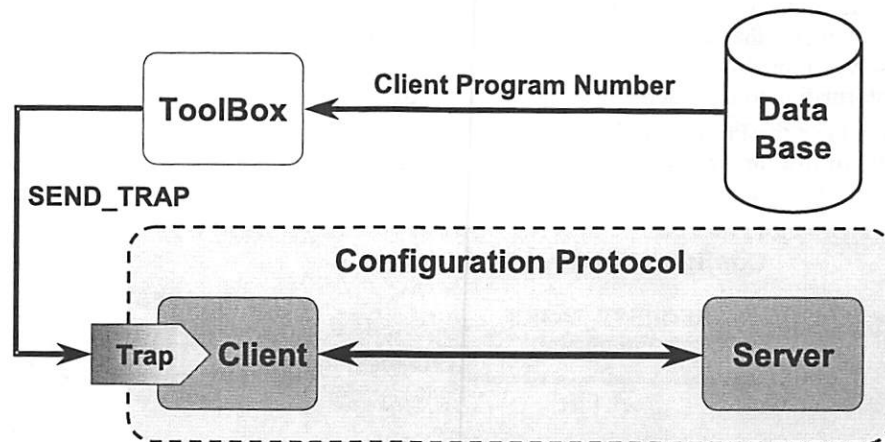


Figure 7: Send Trap.

Initially, the ToolBox reads the Client program number from the Database. This number was previously stored when the Client activated the first iteration of the Protocol and called the *Send Program* procedure (Figure 5). Then, the ToolBox calls the *Send Trap* procedure to notify the presence of modifications in the Client's configuration. According with the Client's state, the Protocol is immediately activated or delayed, as discussed above.

Upon the receipt of the trap signal, the Client instantly sends a status code to the ToolBox. This code indicates the current state of the Client: executing the Protocol or waiting the timer of the pull mode. The ToolBox is unblocked when it receives this status code.

The Database Access Interface

The Interface is composed of a set of eleven procedures, directly related to the operations supported by the Protocol. These procedures perform the data representation format conversion between the Protocol and the Database:

- **Open Connection:** opens the connection with the Database. Before accessing the Database, a Server must open a connection with the DBMS.
- **Close Connection:** closes the connection with the Database. After identifying any error, the Server must close the connection with the DBMS.
- **Request Host_Id:** retrieves the Client's identifier from the Database. The Client retrieves its identifier through the *Request Server* procedure (Figure 4).

The other eight procedures of the Interface have names and functionality that are similar to the Protocol's procedures: *Send Program*, *Request Tasks*, *Request Actions*, *Read File*, *Confirm Action*, *Confirm Task*, *Confirm Configuration*, and *Send Error*.

For example, when the Client calls the *Request Tasks* procedure of the Protocol, the Server activates the same procedure of the Interface to retrieve the information from the Database (Figure 8). In this case, the Interface converts the data format between the DBMS and Server representation, and then, the Server sends the information to the Client.

The Interface allows the Protocol to be independent of the DBMS. To use the Protocol with another DBMS, only the Interface has to be changed. The

Interface allows the Server to remain connected to the Database during its execution, without opening a new connection for each request. This behavior enhances the performance of the system.

The Implementation

An implementation was developed to validate and evaluate the system's architecture and the Protocol. This implementation was developed in C language for Sun Solaris 2.5.1. The availability of the current implementation to others platforms depends on the portability of the C language and RPC library.

The design and implementation of the ToolBox, the data model of the Database and the interaction mechanism between ToolBox and Database are not included in this paper. However, to validate the Protocol implementation, these elements (ToolBox and Database) were developed.

The Database was implemented using the O2 object-oriented database management system [13]. The Database has a set of object classes to represent the configuration structure, pointed out previously. The Toolbox was implemented as a set of programs in O2C, a language that belongs to the O2 System. It has been used and tested to perform some tasks to configure the NFS service.

The Protocol was developed using Remote Procedure Call (RPC) [11] and its Client and Server programs were written in C. The Interface was coded in C to access the O2 database and was integrated to the Server.

The implementation has full functionality according to the specification of the Protocol. Future works must include security and access control mechanisms among Clients and Servers. Initially, we have to evaluate the encryption mechanisms provided by RPC platform, for instance, the Data Encryption Standard (DES).

The Client can be protected by defining a new kind of service that identifies the trusted Servers for a given Client. So, when the Client retrieves the list of available Servers through the *Request Server* procedure, it can check if the Servers are reliable. The Server processes any request only when the host, task and action identifiers belong to the host sending the request.

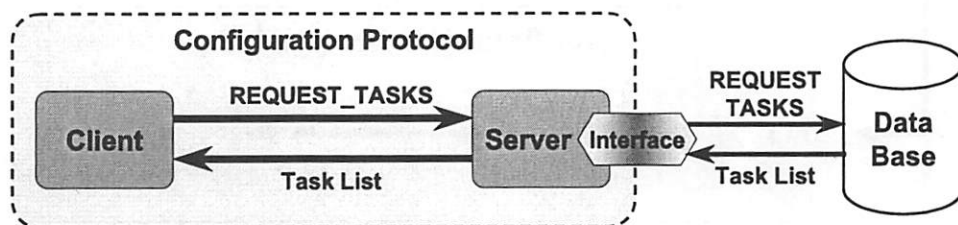


Figure 8: Interface's Procedures.

The broadcast mechanism used by the Protocol facilitates the initialization of the Clients and Servers, but does not allow a Client to use a Server in a different physical network. Multicast mechanism can be used to allow Servers and Clients to be in different networks.

The Server is executed as a single thread. This feature imposes some processing delays in medium-size networks. In large networks, these delays can become noticeable. To eliminate this problem, a new version of the Server can be developed using multi-threads.

Deployment and Use of the System

To use the system, the administrator must install and activate its components appropriately. Figure 9 shows a typical installation and activation of the system.

First, the DBMS used to keep the Database must be activated in the network. In our case, the O2 System must be activated. The Clients do not access the Database directly. They retrieve their configuration information through the Servers. So, the Client hosts do not need execute any software related to the DBMS. On the Server side, when the DBMS supports access over the network, only one host executing the DBMS is needed. The current implementation needs only a single host executing the O2 System.

The next step is to choose the hosts that will act as Servers. There must be at least one Server in each network segment. To obtain higher levels of resilience, it is suggested to install more than one Server in each segment. To minimize traffic on the network and increase the performance of the system, it is suggested to install a Server and the DBMS software on the same host.

Since the Servers are installed, the administrator must install and activate the Client in each host that needs its configuration to be managed. Even the

Servers have to execute the Client code if they need to be managed. This is also valid for hosts executing the DBMS software.

At this point, the system is running and the services can be configured using the ToolBox. As discussed before, different implementations of the ToolBox, with different functionality, can be constructed using the CDS as the configuration propagation and activation platform. In [9], a full-scale configuration management system is described, that uses CDS as the underlying distribution mechanism.

Concluding Remarks

The Configuration Distribution System (CDS) presented in this article has several important features, including:

- **Simplicity:** the action types of the Protocol are simple to implement and understand, avoiding the complexity of specific commands for each service.
- **Flexibility:** the Protocol is sufficiently flexible since it supports arbitrary commands in a sophisticated way.
- **Adaptability:** the set of action types allows any service to be configured on a host.
- **Genericity:** the distribution system is generic and can be used with any other tool that support configuration planning, consistency checking and configuration file generation, as explained above. It can also be used without such system, in which case the configuration files would be manually generated.
- **Stability:** the Protocol is stable since its definition (and therefore its implementation) remains fixed even if new services or platforms need to be added to the system.
- **Performance:** the experiments have shown that the Protocol has an excellent performance, obtained by the simplicity of its operations and incremented by the dynamic distribution of the

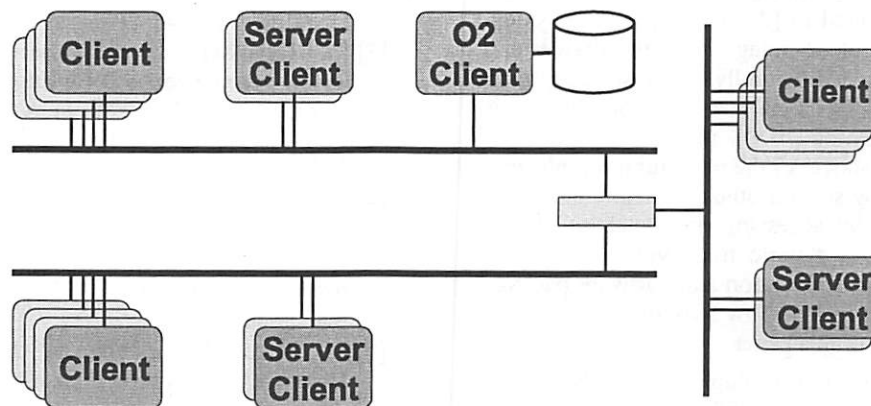


Figure 9: Typical Environment.

processing load on the Servers, that is implicit in the selection process of the Server.

- **Resilience:** the stateless behavior of the Protocol and the possibility of defining several Servers in a given network confer a high level of resilience to the system.
- **Robustness:** the set of facilities pointed out above makes the system robust even in high load situations.

This set of features gives to the system exceptional advantages in large-scale heterogeneous networks:

- **Service and Platform Independence:** the Protocol's operations are suitable to support the configuration of any service across different platforms. The particularities of the services and platforms were abstracted away for the Protocol and inserted in the files and commands manipulated by the actions.
- **Database Independence:** the Interface provides independence from the particular choice of implementation of the Database. To support a new kind of DBMS, only the Interface must be developed. Standard access mechanisms, for instance, ODBC or JDBC, can implement a generic interface, which operates on any DBMS supporting the mechanism.
- **Integration of Administration Tools:** the Protocol allows the integration of administration tools through the Database. In such view, different tools access and store configuration information in a common database, and, the Protocol propagates that information to the hosts, independently of how the configuration has been generated.

The facilities of the Interface can be used to design a new architecture based in multiple Databases with different structures and formats. In such architecture, the Database must store the location of the host's configuration. Prior to process a request, the Server identifies the location of the Client's configuration, and then processes the request.

When compared with existent solutions, in particular those presented in [2,3,4,5,6,7,10], the system presents a number of advantages: a) new services and platforms can be added naturally by constructing their specifications in the Database. b) consistency is greatly improved by the DBMS, instead of flat files, as in all works cited above. c) the configuration information can be used by several others system administration tools merely by accessing the Database. d) the system is, in fact, a generic framework that can be extended to support application and software package management, although to show how this can be done is out of the scope of this paper.

The implementation is stable and has been used to assist the administration of the network of the Department of Informatics, UFPE.

Acknowledgements

The FLASH project is co-funded by the Brazilian Government agency CNPq, through the ProTeM-CC Program (Phase III) and by the Center for Advanced Studies and Systems at Recife (CESAR). Glêdson E. da Silveira receives a scholarship from CAPES.

Availability

The system is currently packaged for distribution and can be retrieved from <http://www.di.ufpe.br/~flash>.

Author Information

Glêdson E. da Silveira is a lecturer of the Department of Informatics at the Federal University of Rio Grande do Norte (DIMAp/UFRN), Brazil. He holds a M.Sc. in Computer Science from the Catholic University (PUC) from Rio de Janeiro, Brazil. Currently, he is a Ph.D. student in the Department of Informatics at the Federal University of Pernambuco (DI/UFPE), Brazil. Reach him electronically at ges@di.ufpe.br.

Fabio Q. B. da Silva is an Associated Professor of the Department of Informatics at the Federal University of Pernambuco, Brazil, where he coordinates the FLASH Project. He holds a Ph.D. in Computer Science from the University of Edinburgh, Scotland. He is also the Finance Director of the Center for Advanced Studies and Systems at Recife (CESAR), a not-for-profit organization dedicated to promote Industry/University interaction (<http://www.cesar.org.br>). Reach him electronically at fabio@di.ufpe.br.

References

- [1] J. S. da Cunha, G. E. Silveira, F. Q. B. da Silva and J. N. de Souza. "An Object-Oriented Service Configuration Management System." *International Conference on Telecommunication (ICT-98)*, Chalkidiki, Greece, June, 1998.
- [2] P. Anderson. "Towards a High-Level Machine Configuration System." *8th USENIX System Administration Conference (LISA VIII)*, San Diego, September, 1995.
- [3] M. Harlander. "Central System Administration in a Heterogeneous UNIX Environment: GeNU-Admin." *8th USENIX System Administration Conference (LISA VIII)*, San Diego, September, 1994.
- [4] J. P. Rouillard and R. B. Martin. "Config: A Mechanism for Installing and Tracking System Configurations." *8th USENIX System Administration Conference (LISA VIII)*, San Diego, September, 1994.
- [5] M. Fisk. "Automating the Administration of Heterogeneous LANs." *10th USENIX System Administration Conference (LISA X)*, Chicago, September, 1996.

- [6] I. Hideyo. "OMNICONF: Making OS Upgrades and Disk Crash Recovery Easier." *8th USENIX System Administration Conference (LISA VIII)*, San Diego, September, 1994.
- [7] Xev Gittler, W. Moore, J. Rambhaskar. "Morgan Stanley's Aurora System: Design a Next Generation Global Production Unix Environment." *9th USENIX Systems Administration Conference (LISA IX)*, 1995.
- [8] Hal Stern, *Managing NIS and NFS*, O'Reilley & Associates Inc., 1991.
- [9] Fabio Q. B. da Silva, Juliana S. da Cunha, Danielle M. Franklin, Luciana S. Varejao and Rosalie B. Belian. "An NFS Configuration and Management System and its Underlying Object-Oriented Model." *12th USENIX System Administration Conference (LISA XII)*, Boston, December, 1998.
- [10] D. Pukatzki e J. Schumann. "AUTOLOAD: The Network Management System." *6th USENIX System Administration Conference (LISA VI)*, Long Beach, October, 1992.
- [11] Sun Microsystems Inc. *RPC: Remote Procedure Call - Protocol Specification V2. RFC 1057*, June, 1988.
- [12] FLASH Project. <<http://www.di.ufpe.br/~flash>>.
- [13] O2 Technology Inc. *The O2 System User Reference*. November, 1995.

An NFS Configuration Management System and its Underlying Object-Oriented Model

Fabio Q. B. da Silva, Juliana Silva da Cunha, Danielle M. Franklin, Luciana S. Varejão, and Rosalie Belian – Federal University of Pernambuco

ABSTRACT

This paper describes an NFS configuration and management system for large and heterogeneous computer environments. It also shows how this system can be extended to address other services in the network. The solution is composed of a process that describes service configuration and management life-cycle, a modular architecture and an objected oriented model. The system supports multiple features, including: automatic host and service installation, service dependency inference and analysis, performance analysis, configuration optimization as well as service functioning monitoring and problem correction.

Introduction

The installation and configuration of hosts and services are two very common tasks in the administration of any computer network. Even small and homogeneous networks offer several different services that must be consistently configured on server and client computers. Furthermore, configuration management becomes a very complex problem when large and heterogeneous systems, together with their Internet connection, are taken into account. According to several authors [2,3], some of the reasons for this increase in complexity are [1]:

- Configuration of each service must be uniform over the network, requiring update on every host anytime a configuration change is required;
- In general, there are great differences in the actual format and location of the configuration files of a service for each platform. Therefore, to configure a service in a heterogeneous network amounts to configure the service in each platform separately;
- Each operating system provides its own set of non-standard configuration parameters for the common network services. In most cases, it is necessary to learn and use these non-standard features to achieve optimal performance of the service in each platform.
- Usually, large networks are managed by a team of system administrators. Configuration rules and parameters must be effectively communicated among team members to avoid inconsistencies that can arise due to personal preferences during the configuration process.
- It is difficult to detect and correct services misfunction before any damage to the users;

- It is also difficult to see dependencies between the services.

The configuration of hosts and services have dramatic influence on network performance, resilience and safety, and therefore are among the most critical tasks in system administration. For instance, it is widely known that a large percentage of security problems on networks connected to the Internet are due to bad Internet services (like HTTP and FTP) configuration.

Therefore, more systematic and structured processes of service configuration are necessary for the administration of today's networks and, in particular, those that are connected to the Internet. Furthermore, to achieve high levels of consistency and safety, any such process must be supported by automated tools which must possess three fundamental properties:

- efficient, robust and user friendly operation on large networks (hundreds of servers and thousands of clients);
- transparent support of heterogeneous platforms;
- consistent support of different services across every supported platform, and the possibility of co-relating them for analysis purposes.

This article presents a service and configuration management system that supports NFS configuration and monitoring on large and heterogeneous networks. The system is based on a formal model that describes hosts, network components and services in a generic and abstract way. This model allows the system to be extended to support several different services in a variety of platforms.

The next section will present the service configuration problem, the related work and how our solution extends the state of the art. Subsequently, we will show the process that describes an NFS service

configuration and management life-cycle. The next section will present the system architecture. Then we will demonstrate the generalization of our solution for the other network services. Finally, we will present our conclusions.

Service Configuration and Related Work

Typically, the configuration of a service is composed of several interconnected tasks: planning, consistency checking, deployment, and management. In each task, a number of critical issues must be addressed, including:

- capacity of servers to offer the required service;
- the placement of the server with respect to the network topology, to avoid network performance problems and bottlenecks;
- dependency relation among hosts, with respect to different services, to avoid many single points of failure on the network;
- configuration consistency on all servers and clients, for ease maintenance and crash recovery;
- documentation of the entire process to allow consistent and efficient administration.

These are only a few of the issues that must be addressed in the process of host and service configuration. Hardware vendor's specific tools, like the AdminTool from Sun Microsystems Inc. and Smit from IBM Inc., do not provide satisfactory solutions to deal with heterogeneous system.

According to Evard [20], "the general approach taken by the administrative community over this time period has been to develop a host cloning process and then to distribute updates directly to hosts from a central repository." In a large and heterogeneous network, this cloning process is not satisfactory because each machine has its own characteristics.

In most solutions, the central repository is a collection of ASCII files, like in LCFG [1], GeNUAdmin [2], Syslogd [4] and Fisk's system [5]. This leads to the problem of keeping the consistency and the integrity of the information. Some tools have changed this approach by using DBMSs, like UHA [14], Aurora [19], Finke's system [13].

The swatch [16] and pong [15] systems are designed to monitor the network and some services. However, neither addresses the complete life-cycle of service configuration and monitoring, and, therefore, does not support integrated service management.

None of the above cited approaches are based on a formal and generic conceptual model of the network and its services. Such a model is the basis of commercial tools like TME10([17] and Unicenter TNG([18], and are essential to support the inclusion of new services and to keep the overall integration of the system. These two market leaders offer extensive features to manage heterogeneous networks, but do not offer built in facilities for high level configuration management.

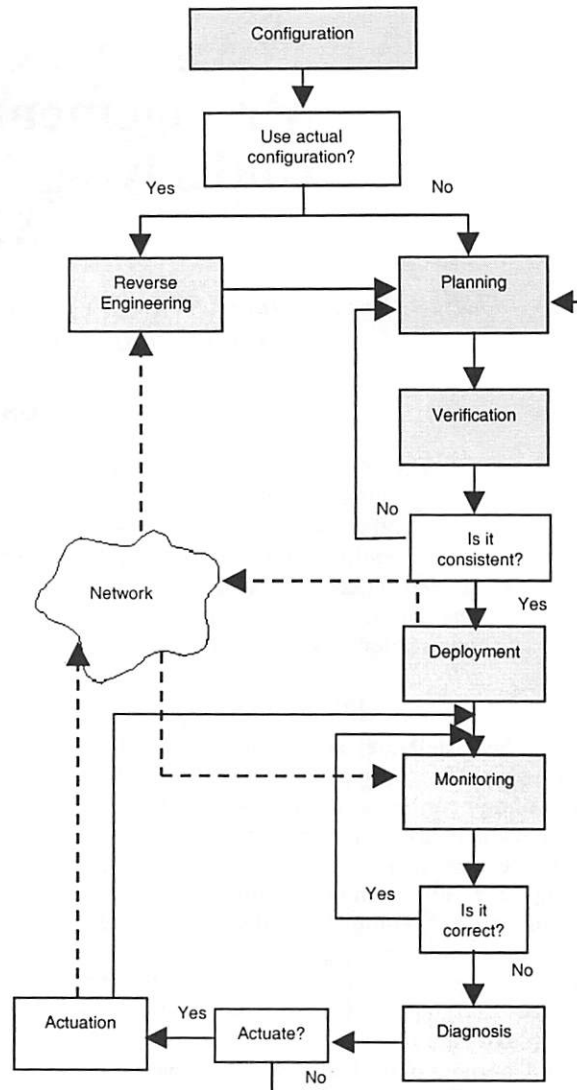


Figure 1: The Process.

In this paper we present a solution dedicated to address the problem of configuring and managing network services in heterogeneous environments. This system presents the following features:

- it allows service configuration to be carried out in an abstracted away from hardware and operating system details;
- service configuration is performed on a central database and is distributed automatically over the network. This enforces consistency and uniformity of the configurations on all hosts that use that service;
- since configuration distribution is performed automatically, the system scales up to deal with large number of hosts;
- it allows new services and platforms to be easily added;
- it supports reconfiguration, service dependency inference and analysis, performance analysis and configuration optimization in a single framework;

- it supports service monitoring, trouble shooting and problem correction.

An NFS Configuration and Management Process

The configuration and management of the NFS service follows a process that is common to most services, and is graphically depicted in Figure 1. Following this process, the administrator should perform a number of tasks in a coherent and consistent form:

- **Planning:** to define the filesystems exported by the servers and imported by the clients, as well as their access and security characteristics. This task should abstract away from platform specific features. In most cases, a service plan is constructed from the configuration information currently in use on the network. To extract this information, a Reverse Engineering stage is necessary. Therefore, the Planning stage in the process can start either with fresh configuration information or with information extracted from the network.
- **Consistency checking:** the planning of the service must be carefully checked for consistency. For instance, in large networks it is fairly easy for the system administrator to lose control over the import and export relationship among hosts. If this happens, servers may end up exporting filesystems that are not used by any client and clients may try to import filesystems that are not exported by the server. Moreover, clients may try to mount a read-write filesystem that is exported as read-only, leading to an error.
- **Deployment:** the deployment of a service configuration is composed of three sub-phases:
 - **Generation of configuration files:** from a consistent service plan, the NFS export and import files should be generated. At this stage, platform specific features must be used to create files on the right format for each supported operating system and hardware platform.
 - **Configuration propagation:** the export and import files must then be distributed to the corresponding hosts.
 - **Configuration activation:** the necessary actions must be performed on each host to activate the new configuration. This involves rebooting the host.
- **Management:** the service must be monitored and controlled during execution, and problems must be identified and corrected. During the Management sub-process, current service behavior is compared to the planned behavior described in the configuration specification. Any deviation is detected and actions are fired to bring the service back to normal operation. If necessary, system administrator is notified for action.

The Architecture of the System

The NFS Configuration and Management System was implemented according to the architecture showed in Figure 2. This architecture was presented by Franklin in [11].

The architecture implements the process showed in Figure 1, which is based in the following stages: planning, deployment, monitoring, diagnosis and actuation.

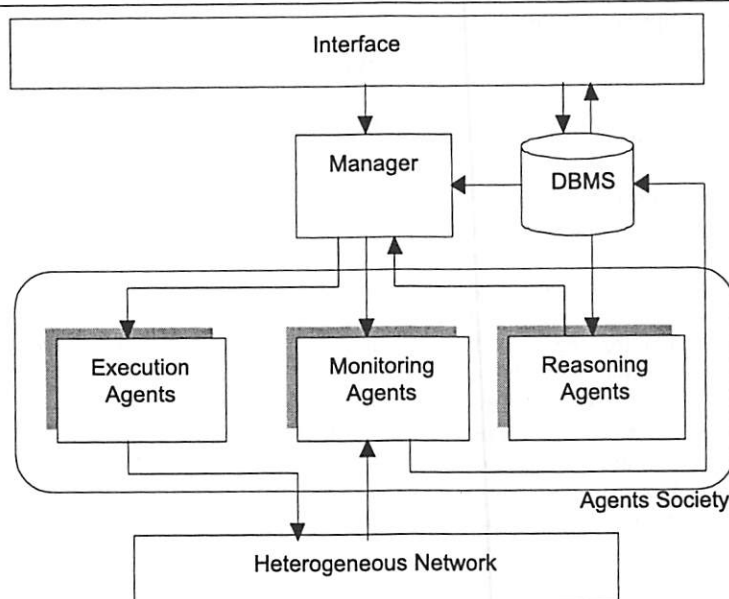


Figure 2: The Architecture.

The architecture has four components: the DBMS, the Agents Society, the Manager and the Interface.

The DBMS

The DBMS is the central element of the architecture and it is partitioned into two components:

- The **Configuration Component**: this is the implementation of the conceptual model described later on. It stores the NFS configuration definition and other information about its behavior in the network. The information resulting from the planning stage is stored in the DBMS using the Interface.
- The **Monitoring Component**: this has two functionalities: (1) it mirrors the configuration component, keeping monitoring information obtained from the network; (2) it supports the monitoring process, keeping monitoring parameters like, for instance, monitoring time intervals and agents status. The information is updated in the monitoring component by the monitoring agents, either as a result of the monitoring or the reverse engineering processes. This component is fully described in [21].

The configuration and monitoring components of the database have been implemented in ORACLE RDBMS.

Agents Society

The **Agents Society**, based on the intelligent agent paradigm described in [12], is composed of three types of agents, with complementary functionalities:

- The **Monitoring Agents** perceive the environment and collect information about the actual behavior of the NFS service on the network. There are two kinds of monitoring agents for NFS: one that searches configuration files and another that verifies the running daemons. These agents travel from the monitoring host to the target hosts, execute the scripts that collect information, receive the result from the scripts and deliver to the manager. The latter feeds the information to the monitoring database. For example, they collect information about which filesystems are imported and exported by the hosts. The monitoring agents are responsible for the Monitoring Stage in Figure 1.
- The **Reasoning Agents** compare the actual behavior of the service and the planned behavior stored in the DBMS, looking for inconsistencies. Based on predefined rules, these agents infer the actions that must be executed to correct possible problems. For example, they verify whether the filesystems imported by the client host are correctly exported by the corresponding server host. Currently, the agents provide the system administrator with a possible action, and he/she is responsible for its

execution. Examples of some of the rules used by the reasoning agents are presented below:

- If imported filesystems are not correctly exported by the corresponding servers, then either servers or clients must be reconfigured.
- If the server's daemons are not running, then start them.
- If there is a difference between the configuration and monitoring databases, then the system must be reconfigured in order to maintain the consistency between network and planning.
- If a client can't mount a filesystem, then verify the server's status and inform the administrator.

These informal rules are coded into the system as sets of formal rules that may activate one or more actions.

The reasoning agents implement the diagnosis stage in Figure 1. The automatic execution of actions by the agents (without system administration intervention) and learning capabilities are under development using the IBM ABE [22] and the knowledge base construction language KIF [23].

- The **Execution Agents** perform actions on the network. That is, they change the configuration on the hosts in the network to eliminate a problem in the NFS functioning, according to what was defined by the Diagnosis Agent, or only to modify the actual configuration of the NFS service. For example, it corrects the inconsistency between the imported and exported files from clients and servers. This agent implements the deployment and actuation stages in Figure 1.

The agents have been implemented in Java, using the IBM Aglets Workbench API [24]. This API supports the construction of mobile multi-agents systems.

The Manager

The **Manager** controls the agents and interaction process between the architecture modules, activating, deactivating and creating the agents when necessary. When a set of agents is activated, it is not necessary to wait for all agents to finish their tasks to record information on the database. The manager records incomplete information and controls the time-out interval (defined by the system administrator) of all running agents. Incomplete information is dealt with by the reasoning agents. The manager has been implemented using the same technology as the agents.

Interface

The **Interface** allows the interaction between the system and the administrators. It provides the following functionalities: the DBMS front-end, visualization of all management processes, the NFS status in the network and visualization of the service planning.

The system interface has been developed using HTML embedded into ORACLE PL/SQL stored procedures. The use of HTML pages allows greater portability and easy access through any web browser. Furthermore, the use of stored procedures facilitates the interaction with the database. However, it makes the interface dependent on the ORACLE database. A more portable solution using JAVA and Servlets to communicate with the DBMS is under construction.

The planning and verification stages are also implemented in PL/SQL and accessed directly through the interface. The remaining stages are implemented by the agents society, as described above.

Communication Protocols

A set of communication protocols are needed: between the agents of the same category, between agents of distinct categories, between the agents and

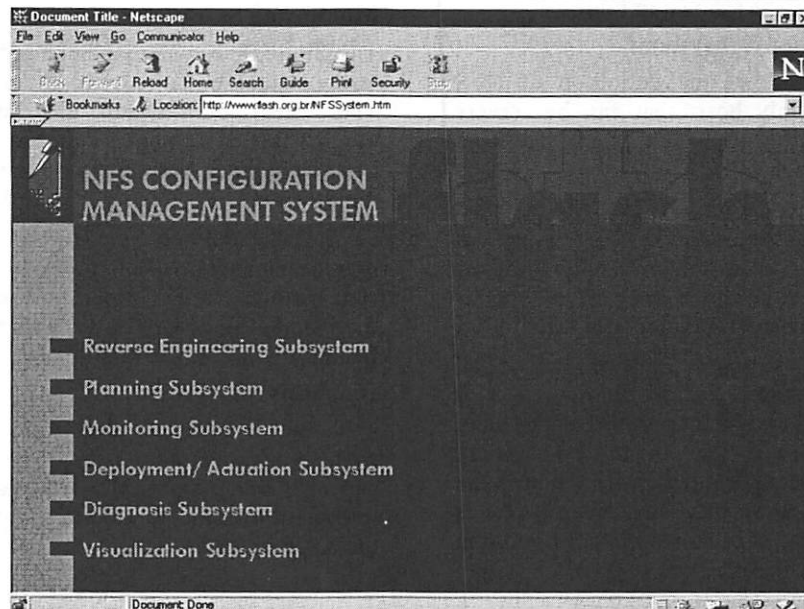


Figure 3: NFS configuration management systems interface.

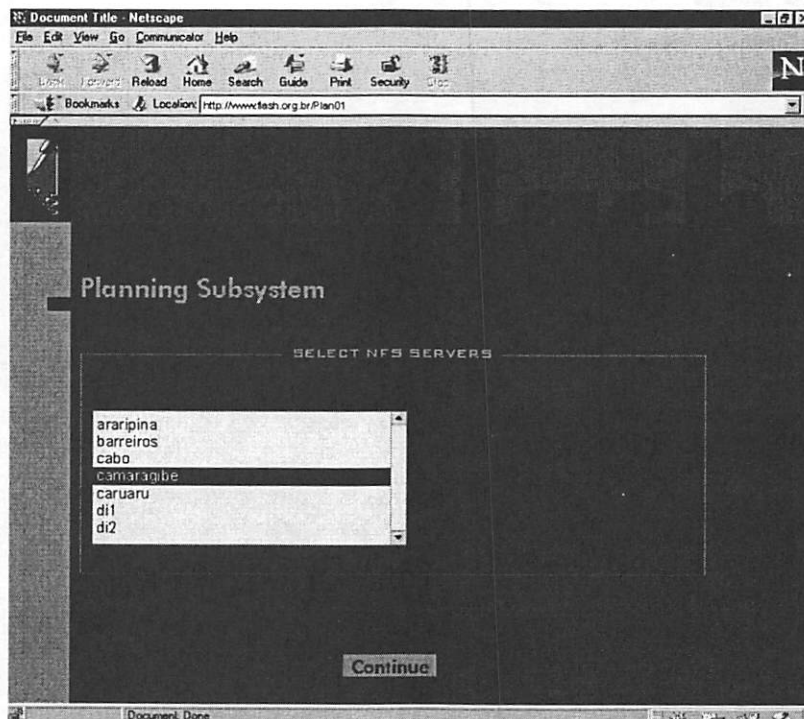


Figure 4: Planning subsystem.

the DBMS, between the agents and the network components. The latter implements the propagation and activation phases of the process described in Section III and is the subject of another paper [9].

The Use of the System

The navigational model follows the cycle defined in Figure 1. Each stage has been implemented as a separate subsystem, and are reached through the first page of the interface, as shown in Figure 3.

Planning and Validation Subsystems

This subsystem allows the definition of NFS clients and servers. Only hosts that can be servers appear in the list box of Figure 4 (e.g., those that have disks). Similarly for clients.

The following step is to configure the selected server and client hosts. As shown in Figure 5, once a host running Solaris 2.5.1 is selected, only the platform specific parameters for this version of the operating system appear in the configuration interface.

Once the NFS planning is finished, that is, all servers and clients are selected and configured, it is necessary to check and validate the configuration. This checking is performed by the validation subsystem, which is a PL/SQL procedure that either requests the system administrator to modify any inconsistencies found in the configuration or stores the configuration in the database if it is correct.

Deployment and Actuation Subsystems

To deploy or modify an NFS configuration, the manager verifies what needs to be updated (a deployment or modification may only affect a subset of the hosts) and schedules the process. The start of the process can be immediate or in a pre-schedule time defined by the administrator, as shown in Figure 6. At the deployment time, the agents and the configuration protocol are activated.

Monitoring Subsystem

In this subsystem, the manager queries the monitoring database looking for monitoring information. The system administrator tells the manager, through the interface, which agents need to be activated, which hosts to monitor and the time intervals of the monitoring processes, as shown in Figure 7.

At the defined time, the manager activates the monitoring agents, which travel to the target hosts and start the monitoring scripts, as discussed above. Once the monitoring process is finished the manager inserts the information in the monitoring database.

Diagnosis Subsystem

The diagnosis is also started through the manager, by a system administrator's request. However, it can also be automatically triggered every time new information from the monitoring process is stored in the database.

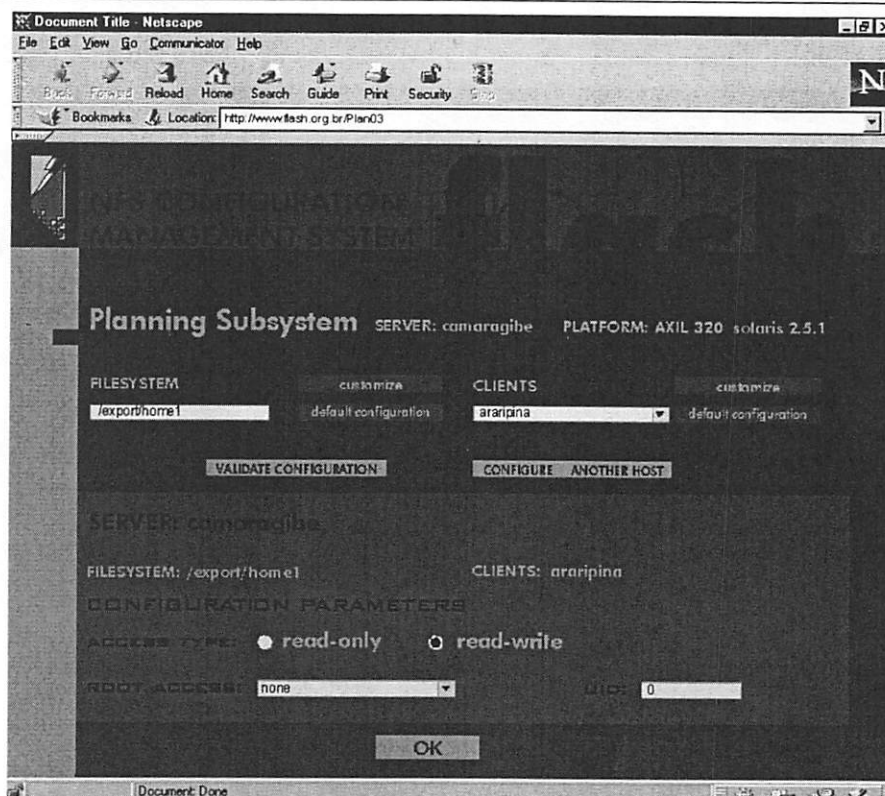


Figure 5: Planning subsystem – second interface.

The manager starts the diagnosis agents which will compare the configuration and monitoring database, looking for possible inconsistencies, and will notify the manager with two possible results:

- an OK sign, indicating that no inconsistencies

have been found;

- the failure points and possible solutions and actions to bring the network back to a consistent state.

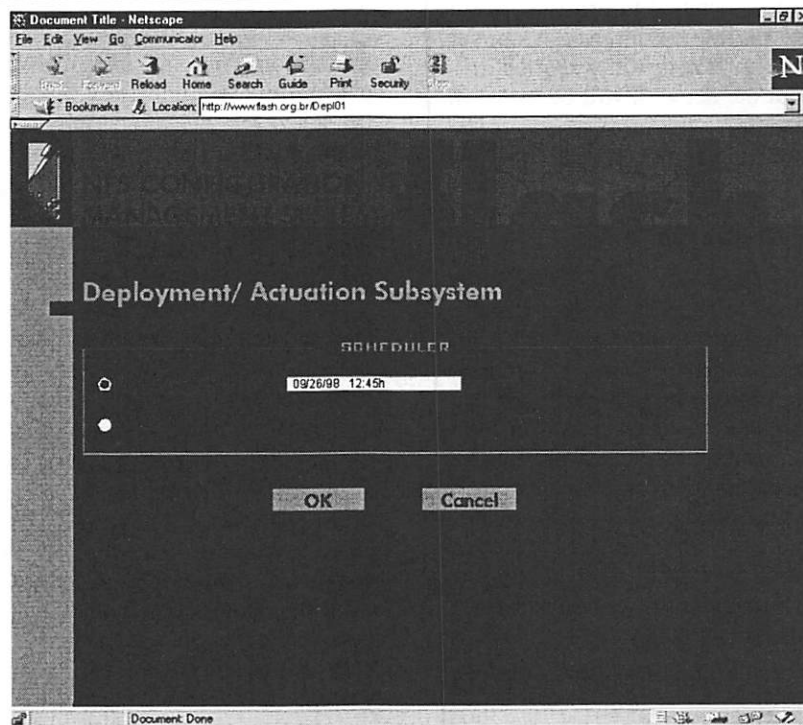


Figure 6: Deployment/actuation subsystem.

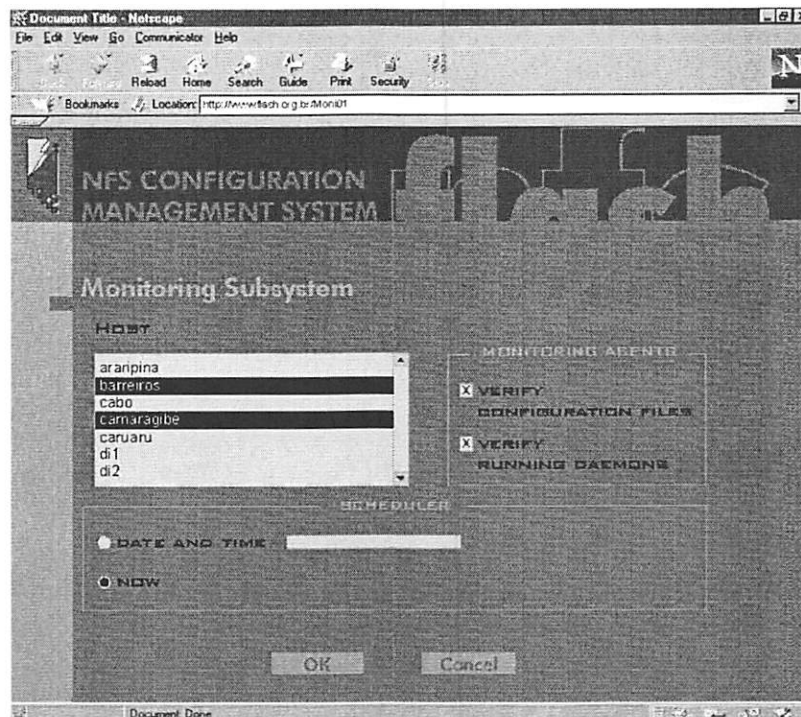


Figure 7: Monitoring subsystem.

The manager is responsible to format the result and send messages to the system administrator.

Reverse Engineering Subsystem

This subsystem is implemented by the monitoring agents already defined and by specific agents capable of collecting more detailed information about the network hosts. The reverse engineering is similar to the monitoring process, but in this case the collected information about the network will be stored in the configuration database. This process has to be authorized by the system administrator, since it changes the network configuration.

Generalization

The generalization depends on a conceptual model, specially designed to support all the necessary components and characteristics to configure and manage services in an heterogeneous network. This model, described in [10], uses object oriented paradigm that allows abstract descriptions of hosts, platform, services and configuration parameters, and dependencies among them. In this model, it is possible to construct a platform independent specification of a given service, which when combined with the specification of a given platform, produces the instantiation of the service for that platform. The inclusion of a new service is easily performed only by class instantiation.

The model showed in Figure 8 has seven classes, described as follows.

The class **Service** contains the description of the service (e.g., filesystem sharing services). This class is useful when there are several products that implements the same service. In this case, each implementation is modeled as a separate product and all versions belong to the same service.

The class **Product** contains product specific information, like, for instance, version and supported platforms (e.g., SunOS 4.1 version of NFS). This class has a self relationship permitting the dependency representation among services.

Each instance of the class **Parameter** represents a configuration parameter (e.g., filesystem access type). New parameters can be added through class instantiation. Each instance of the class **Product** refers to one or more instances of the class **Parameter**, therefore representing the information that is necessary to install product of a given service on the network hosts.

Each configuration parameter associated with a service has a set of restrictions. Different implementations of a service for distinct platforms have different set of restrictions. The restrictions are captured by the class **Parameter Restriction**, where each parameter connected to a service and a platform has well defined rules for its correct use during the configuration process (e.g., the filesystem access type parameter can only assume the values *ro* – read-only or *rw* – read-write).

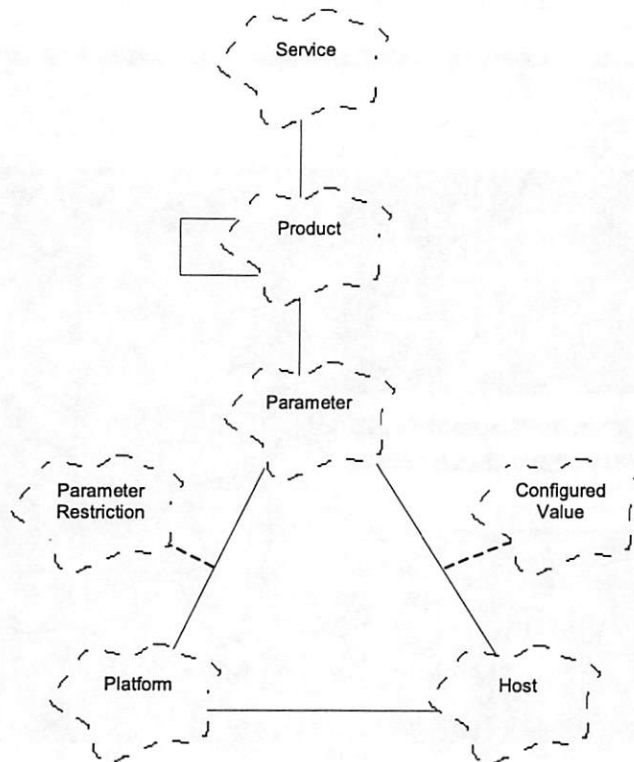


Figure 8: The conceptual model.

The class **Configured Value** holds the actual configuration values for each parameter of a given host (e.g., on host *host1*, the filesystem access type parameter has value *ro* to *filesystem1*). The class **Host** represents all hosts of a computer network and each instance of this class represents an actual host. (e.g., *host1*).

The class **Platform** captures the heterogeneous features of the hardware and operating system platforms on the networks (e.g., host IBM RS/6000 running AIX 4.1). A new platform can be easily added through class instantiation.

This model is independent of any implementation decision. So, it can be implemented in ASCII files or in any desired DBMS. In our case, we used ORACLE DBMS.

This object-oriented conceptual model specifies the database that holds network configuration information, it represents the DBMS component of the architecture showed on Figure 2. From this database, the configuration information is distributed to every host on the network using a distribution protocol. This protocol is described in [9].

Some features offered by this model are listed as follows:

- network service configuration planning;
- support to include new services and related network components;
- support of heterogeneous network;
- consistent and integrated view of the network;
- creation, alteration and visualization of host configuration even if it is down;
- visualization of the service topology, that is, the relations between servers and clients of the same service, and also the visualization and analysis of the services dependency graph, showing the relations among services;
- support to the monitoring and diagnosis phases to avoid failures, redundancies, inefficiencies and inconsistencies.

Conclusions

The service configuration and management system presented in this article has several important and desirable features:

- Consistency and uniformity of the configurations: achieved through the verification of configuration information stored in the database, according to rules also coded in the system.
- Automatic host and service reconfiguration: enforced by the configuration stored in the database that stays available even when the host is down.
- Scalability to deal with large networks: accomplished by the possibility of configuration replication and automatic propagation of configuration information to an arbitrary number of target hosts.

- Easy inclusion of new services and platforms: supported by the capability of defining meta-level configuration information in the database, as described in the previous section.

The solution presented in this article, uses several technologies from the design and implementation of a complex system, capable of support the planning, configuration and pro-active management of services in heterogeneous networks. The NFS prototype was used to validate the process, architecture and conceptual model.

The possibility of supporting other services, coherently integrated in the framework, is a clear advantage of this solution when compared to managing the services isolated and by hand. Currently, HTTP, NIS and DNS services are being included in the system. This will allow not only the configuration management of these systems separately, but also the correlation of information among all supported services.

Acknowledgements

The FLASH project is co-funded by the Brazilian Government agency CNPq, through the ProTeM-CC Program (Phase III) and by the Center for Advanced Studies and Systems at Recife (CESAR).

Availability

The prototype implementation, together with a configuration distribution system [9], has been used to assist the administration of laboratories of the Department of Informatics, UFPE. However, the system will only be available for distribution after the inclusion of the above mentioned services.

Author Information

Fabio Q. B. da Silva is an Associated Professor of the Department of Informatics at the Federal University of Pernambuco, Brazil, where he coordinates the FLASH Project. He holds a Ph.D. in Computer Science from the University of Edinburg, Scotland. He is also the Finance Director of the Center for Advanced Studies and Systems at Recife (CESAR), a not-for-profit organization dedicated to promote Industry/University interaction (<http://www.cesar.org.br>). Reach him electronically at fabio@di.ufpe.br.

Juliana Silva da Cunha is a Phd Student of the Department of Informatics at the Federal University of Pernambuco, Brazil and a member of The FLASH Team. She holds a master degree in Computer Science at Federal University of Ceará, Brazil. You can reach Juliana electronically at jsc@di.ufpe.br.

Danielle M. Franklin is currently a member of The FLASH Team and a system management consultant at CESAR. She holds a master degree in Computer Science. You can reach Danielle at dmf@di.ufpe.br.

Luciana S. Varejao is currently a master student in Federal University of Pernambuco's Informatics Department and a member of The FLASH Team. She holds a bachelors degree in Electronic Engineering. You can reach Luciana at lsv@di.ufpe.br.

Rosalie Belian is a member of The FLASH Team. She holds a Master Degree in Computer Science. You can reach Rosalie electronically at rbb@di.ufpe.br.

References

- [1] Paul Anderson, "Towards a High-Level Machine Configuration System," *USENIX Systems Administration (LISA VIII) Conference Proceedings*, 1994.
- [2] Magnus Harlander, "Heterogeneous Unix Environment: GeNUAdmin," *USENIX Systems Administration (LISA VIII) Conference Proceedings*, 1994.
- [3] John Rouillard and Richard Martim, "Config: A Mechanism for Installing and Tracking System Configurations," *USENIX Systems Administration (LISA VIII) Conference Proceedings*, 1994.
- [4] Rex Walters, "Tracking Hardware Configuration in a Heterogeneous Network with syslogd," *USENIX Systems Administration (LISA IX) Conference Proceedings*, 1995.
- [5] Michael Fisk, "Automating the Administration of Heterogeneous LANs," *USENIX Systems Administration (LISA X) Conference Proceedings*, 1996.
- [6] Grady Booch, *Object-Oriented Analysis and Design With Applications, Second Edition*, Addison-Wesley, 1994.
- [7] FLASH Project, <<http://www.di.ufpe.br/~flash>>
- [8] Hal Stern, *Managing NIS and NFS*, O'Reilly & Associates Inc., 1991.
- [9] Glêdson E. da Silveira and Fabio Q. B. da Silva, "A Configuration Distributed System for Heterogeneous Networks," *USENIX Systems Administration (LISA XII) Conference*, 1998.
- [10] Juliana Silva da Cunha, Glêdson E. da Silveira, Fabio Q. B. da Silva, J. Neuman de Souza, "An Object-Oriented Service Configuration Management System," *International Conference on Telecommunication (ICT-98)*, Chalkidiki, Greece, June 1998.
- [11] Danielle Franklin. *I-DREAM: an Intranet based Resource and Application Monitoring system*. Master Degree Thesis, Federal University of Pernambuco, 1997.
- [12] Michel Wooldridge, Nicholas R. Jennings, "Intelligent Agents: Theory and Practice," *Knowledge Engineering Review*, Cambridge University Press, 1995.
- [13] Jon Finke, "Automation of Site Configuration Management," *USENIX Systems Administration (LISA XI) Conference Proceedings*, 1997.
- [14] Gregory Thomas, James Schroeder, Merilee Orcutt, Desiree Johnson, Jeffrey Simmelink, John Moore, "UNIX Host Administration in a Heterogeneous Distributed Computing Environment," *USENIX Systems Administration (LISA X) Conference Proceedings*, 1996.
- [15] Helen Harrison, Mike Mitchell, Michael Shaddock, "Pong: A flexible network services monitoring systems," *USENIX Systems Administration (LISA VIII) Conference Proceedings*, 1994.
- [16] Stephen Hansen, E. Todd Atkins, "Automated system monitoring and notification with swatch," *USENIX Systems Administration (LISA VII) Conference Proceedings*, 1993.
- [17] Tivoli Systems Inc., <http://www.tivoli.com>.
- [18] Computer Associates Inc., <http://www.cai.com>.
- [19] Xev Gittler, W. Moore, J. Rambhaskar, "Morgan Stanley's Aurora System: Design a Next Generation Global Production Unix Environment," *USENIX Systems Administration (LISA IX) Conference Proceedings*, 1995.
- [20] Rémy Evard, "An Analysis of UNIX System Configuration," *USENIX Systems Administration (LISA XI) Conference Proceedings*, 1997.
- [21] Luciana S. Varejao. *Sistema de Monitoração para Redes Heterogêneas (A Monitoring System for Heterogeneous Networks)*. Master Degree Thesis (under development), Federal University of Pernambuco, 1998.
- [22] Agent Building Environment, <http://www.networking.ibm.com/iag/iagsoft.htm>.
- [23] Knowledge Interchange Format (KIF) <http://logic.stanford.edu/kif/>.
- [24] IBM Aglets Workbench Homepage <http://www.trl.ibm.co.jp/aglets/>.

Design and Implementation of an Administration System for Distributed Web Server

C. S. Yang and M. Y. Luo – National Sun Yat-Sen University, Taiwan, R.O.C.

ABSTRACT

The explosive growth of the World Wide Web has raised great concerns regarding many challenges – performance, scalability and availability of the Web system. Consequently, Web site builders are increasingly to construct their Web servers as distributed system for solving these problems, and this trend is likely to accelerate. In such systems, a group of loosely-coupled hosts will work together to serve as a single virtual server. Although the distributed server can provide compelling performance and accommodate the growth of web traffic, it inevitably increases the complexity of system administration. In this paper, we exploit the advantages of Java to design and implement an administration system for addressing this challenging problem.

Introduction

The explosive growth of the World Wide Web (WWW for short) [1] has raised great concerns regarding many challenges – performance, scalability and availability of the Web system. Due to the exponential growth of the World Wide Web, many popular Web sites are being overwhelmed by huge requests and suffer from server overload. In order to cope with the continued increasing demand on their Web servers, Web site managers must continually increase the server's capacity for providing the desired levels of quality of service. Upgrading the server to a more powerful machine may only solve the problem in the short term and might not be cost-effective in the long term. An increasingly popular solution for these problems is to deploy a set of computers, and enable them to work together as a single virtual server. A Web server based on such approach can provide the scalability necessary to keep up with growing client demand at popular Web sites. It could allow additional processing power to be dynamically added to increase the total capacity of the server as demand grows. The other key advantage of such approach is that one can more easily build a server that can tolerate hardware or software failures.

However, although such distributed server can provide compelling performance and accommodate the growth of web traffic, it inevitably increases the complexity of system administration. Failures, performance inefficiencies, content management, resource allocation, security compromises, and accounting are some of the problems associated with the operation of traditional server. When the server is composed of a group of loosely-coupled machines, the administrative burden will be larger. Reliable operational status of such distributed server, unless made in an automated way, requires significant human effort for propagating

one administrative function to all nodes. In particular, such distributed servers tend to be more heterogeneous, and this heterogeneity will come at the cost of greatly increased management costs. As a result, effective management mechanisms and tools are required to mask the complexity and heterogeneity of internal composition of such distributed server, and manage the server composed of several nodes as easy as manage a single node. In this paper, we describe the work we are pursuing in exploiting the advantages of Java [2] to design and implement an administration system for addressing these problems. With the innovative administration system, the Web site manager can manage and maintain the distributed server as a single large system. In addition, the administration system running in the background can also make the clustered server a better place to work.

The rest of this paper is organized as follows. The next gives an overview of distributed Web server. Subsequently, we describe the management problems raised by such an environment, and then present an overview of the architecture of proposed system. The details of implementation are given in the next section. Then, we discuss some related issues and compare our system with other related works, before presenting the conclusion and future work.

Overview of Distributed Web Server

With the rapid growth of the Internet, and the Web in particular, it is difficult for organizations and people running web sites to predict future demands that clients will place on their servers. Consequently, the Web server should be designed to be capable of meeting the evolving demand constantly and easily. That is, if the offered load begins to exceed the server's capabilities, there should be an easy way to scale up the system when the hardware or software of

the existing server does not need to be replaced. In addition, as more and more commercial applications appear on the WWW, it has become apparent that the function performed by these servers is critical. Consequently, making these servers highly available is another problem that is gradually getting attention.

For addressing these problems, we [3,4,5] designed and implemented a scalable and highly available Web server. Figure 1 illustrates the overview of our system. The entire server is composed of a group of interconnected machines. For providing the illusion of single virtual server across these machines, the entire servers cluster should be presented to the outside world by single addressing interface (e.g., single Domain name). Consequently, we designed and implemented a distributing mechanism to spread all incoming Web requests destined for this addressing interface. Such a mechanism contains the following three functions: load balancing, failure detection, and directing. That is, when a new HTTP request arrives, some load balancing mechanisms are invoked for selecting a suitable node to serve the request, and such a mechanism could ensure that the workload is evenly spread

among these server nodes. If one server node goes down, or during periods of maintenance, the distributing mechanism could discover it and respond by directing new requests to the other available nodes. Finally, a directing mechanism is required for directing the request to the selected node, in a manner that is transparently to the outside user. We [3,4] analyzed the TCP/IP [6,7] and HTTP [8,9] protocols that the web is based on, and we found out a number of ways in which one could direct requests to a selected node. After evaluating the tradeoff among these methods, we [4,5] choose reroute TCP connection as the directing method in our system. We extend the Linux kernel to build in all mechanisms mentioned above for fulfilling the distributor, which performs the distributing mechanism in the distributed server.

One node in the system will run the modified kernel to serve as distributor for distributing incoming web requests; the other nodes will execute Web server software responding the incoming HTTP request. Each host participating in the clustered server has a unique IP address, but only the distributor's IP address is associated with the domain name representing this

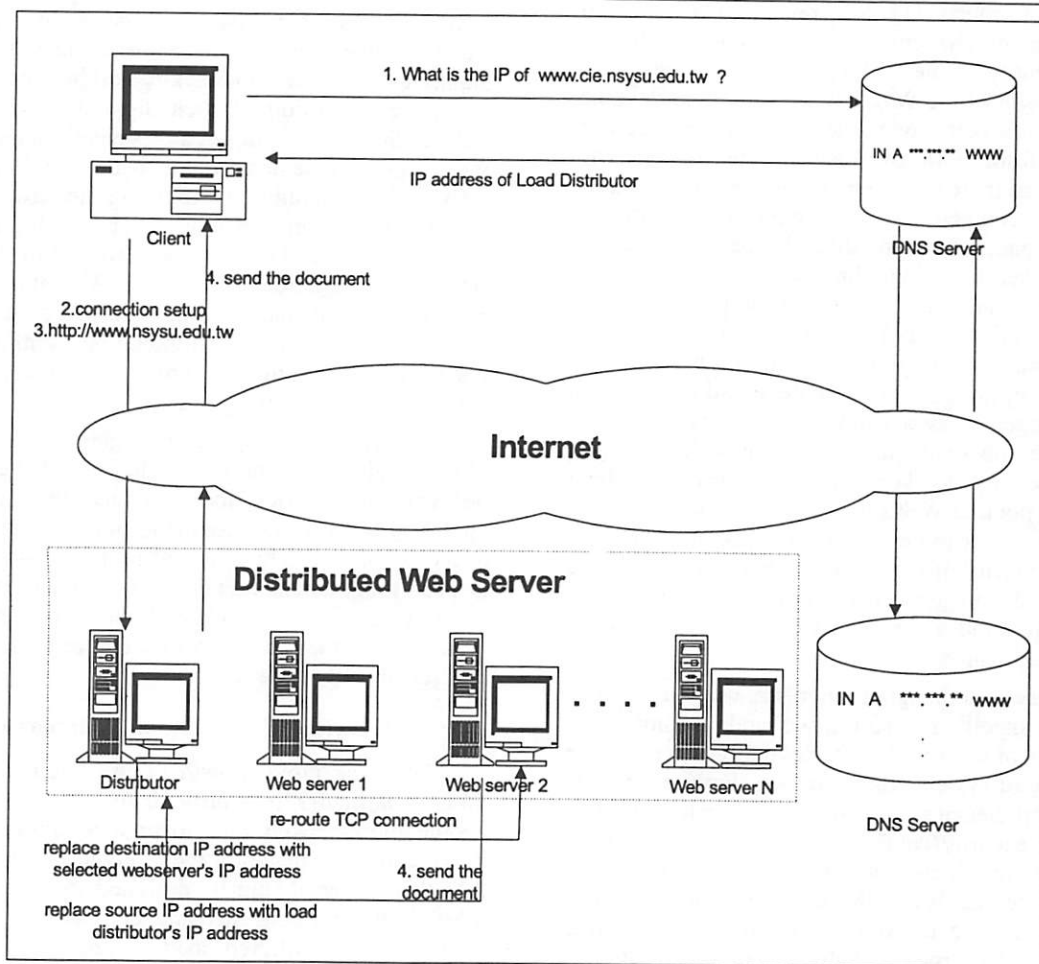


Figure 1: Overview of Distributed Web Server.

web site. As a result, only the distributor's IP address is advertised to the outside world, so that all HTTP requests destined for this web site will be delivered to distributor. The distributor will forward these incoming requests to the selected nodes via distributing mechanism.

In addition, as the incoming requests may be distributed to any node in the system, we also must provide a mechanism to ensure that each server node would look and feel identical to the requesting clients. In other words, we must make each node with the same capability of responding to requests for any portion of resource that the web site provides. For solving this problem, all content provided by the web site can be served from a centralized network file system. However, accessing data over the network file system can be very slow due to the overhead of LAN congestion. Furthermore, such a design will suffer from the single-point-failure problem, which will make the entire system more vulnerable. As a result, we choose an alternative approach as the content sharing method in our system: replicating all content on the local file system of each server nodes.

With these mechanisms, although all machines participating in the servers cluster are autonomous, they will seamlessly work together to serve the requests and provide the illusion of a single server. The new machine can be dynamically added to increase the total capacity of the server as demand grows. We expect that this approach should be attractive, because it can preserve the previous financial investment and reduce the costs of scaling the server. The other key advantage of this architecture is that we can more easily build a server that can tolerate hardware or software failures.

Although the foregoing system can provide a degree of high availability, the failure of distributor will still bring down the entire web server. We designed a primary-backup mechanism for addressing such a problem. One backup distributor can be setup to prevent the problem of single point failure. The primary distributor will broadcast a message periodically in addition to perform the distributing mechanism. The backup distributor treats the message as "heart-beat" of the primary distributor. It performs the same mechanism as just mentioned for detecting the failure of primary distributor, and takes over the responsibility of distributing request when the primary distributor fails.

Proposed System

Design Consideration

Although this system can accommodate growth of web traffic, it inevitably increases the complexity of system administration. Unlike traditional single-server configuration in which the web site manager can has full control on the whole system easily. When the entire Web server is composed by a group of loosely-

coupled machines, the administrative burden of managing and maintaining the entire system will be considerable. Consequently, we intend to provide an administrative mechanism for web site manager to mask the complexity and heterogeneity of internal composition of the distributed server, and manage the server composed of several nodes as easy as managing a single node. Such a mechanism should address the following problems to which an administrator of distributed Web server would like to have answers:

- **Content management.** To guarantee that any request could access any resource regardless of the server to which that it is directed, we replicate all content on the local file system of each server nodes. However, the content of a Web site may change over time. When a change occurs, the system must propagate that change throughout the entire Web site. Usually, such changes need to be updated by manager manually. To address the problem, the proposed administrative mechanism should enable the web site manager to be capable of performing functions on all nodes at once.
- **Configuration.** We should provide a mechanism for administrator to know which node is operating as a part of the system. In addition, adding or removing a node should be an easy way and not require the extensive reconfiguration of all other nodes.
- **Self-diagnose.** If any failure or specific condition occurs in the system, the system should automatically inform the administrator by E-mail or other ways. Otherwise, the troubleshooting will become administrator's nightmare when they face such a complex system.
- **Performance and health monitoring.** We should provide a mechanism for administrator to monitor the status (e.g., resource utilization) of each node, ensure the resources provided by the web site are operational, and specified content can be delivered.
- **Security Concerns.** We should provide a mechanism (i.e., watch the log files created by each server node) to identify security problems or other situations.

System Architecture

To fulfill the administration system, we intended to construct a group of daemons running on each node and enable them to cooperate for performing the administrative functions. However, many problems arose when we tried to implement such an idea. The first major problem is platform heterogeneity, which arises from the fact that we hope the proposed system is flexible enough that each server node can use any kind of hardware, operating system, and Web server software. This means that these daemons that make up the administration system must be capable of running on different platforms. The second considerable problem is extensibility (or versioning and distribution

problem). That is, the functionality of this administration system cannot be extended or modified without rewriting, recompilation, reinstallation, and re-instantiation of all existing daemons.

For tackling these problems, we decided to construct the administration system by Java [2,10]. Java is developed to support applications in a heterogeneous environment, which requires that applications be capable of executing on a variety of hardware architectures and operating system. This is achieved by generating an intermediate code called bytecode, which is an architecturally neutral format designed to be transported easily to multiple hardware and software platform. Such a design can relieve both the developer and user of concerns related to heterogeneity of the target platforms. Thus it is the primary reason that we choose Java to implement the administration system. In addition, the another attractive feature of Java is the notion of downloaded executable content (data that contain programs that are executed upon receipt). Base on this, we can implement each administration function in the form of Java class instead of implementing it in the daemon. All daemons distributed on each node will download the appropriate classes from a central location to perform the management task. Such a design will avoid the software distribution

problem. If a new capability needs to be added, all we have to do is to implement a new Java class. We expect that using the capability of downloadable code from Java should provide unlimited possibility to enhance the function of the administration system.

The Java-based administration system is composed of the following four key components: controller, broker, agent, and remote console. Figure 2 illustrates the overall architecture of the proposed system. The broker will run on each Web server node to perform the administrative function, and monitor the status of the managed node. The administrative functions will be implement in agent, which is in the form of Java class. One special daemon called controller will be responsible for receiving request from administrator, and then invokes brokers to perform the delegated tasks by dispatching the corresponding agent. The remote console is a Java applet, which can be run on any Java-enabled Web browser. The administrator can download the remote console and interact with it to perform management operations.

Implementation

In this section we describe the implementation details and the current status of the proposed system.

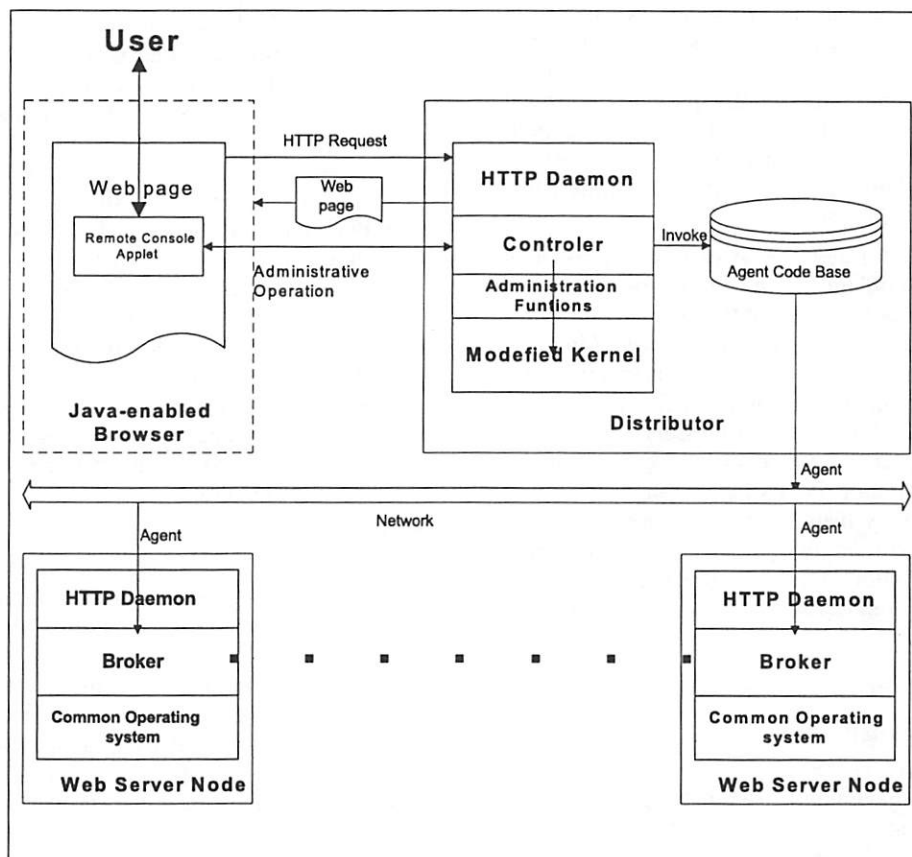


Figure 2: Overall Architecture of the Proposed System.

Control Interface

Because we made the distributor the control center of system administration, as well as distributing requests, we provided several administrative functions for the system administrator to control the system such as the following:

- Turn on/off the distributing mechanism.
- Join/remove a server node into/from the system.
- Read the related statistical data, such as outstanding request of each server nodes.

These functions are user-level programs written in C language. Because all related data are located in the kernel, we also implemented several new system calls that provide an interface for these user-level routines to access kernel-level data.

Controller

The controller is a standalone [11] Java application, which runs as a background process on distributor as the control center of system administration. The main function of controller is to respond the request from system administrator. When the controller starts as a process under the kernel, it forks a main thread to register a socket on a pre-assigned TCP port and block itself on listening mode for any incoming request. Upon a request arriving, the main thread will create a new working thread to carry out the requests so that it can handle other requests waiting in queue or wait for new requests. We expect such multithreaded implementation will allow controller to be capable of serving multiple requests simultaneously with improved response time and throughput.

The request issued by an administrator may be mainly classified into two categories: (1) configuring the system (e.g., add/remove a node into/from the system), or (2) performing a management function (e.g., the web site manager intend to delete or add a file) on all nodes. For condition (1), the working thread will invoke the administrative functions described in the control interface. These administrative functions perform in the form of native method. The native method is a mechanism in Java, which is used to call function that is written in languages other than Java. The administrative function will look up the kernel and setup the related data via the corresponding system call, which is defined in the control interface. In the condition (2), the controller will dispatch the corresponding agent to each server node for performing delegated task. The working thread will keep on listening for the execution result of the agent and report it to the administrator.

Broker

The broker is also a standalone Java application program, which performs as a daemon process on each server node in the system. The broker is composed of two operating threads: agent thread and monitor thread.

Agent Thread

The agent thread is responsible for accepting the dispatched agent from controller, and executing it for performing the delegated task. As a result, it should be capable of loading code from a variety of resources and provide an environment for executing it. For addressing this, we implemented the following components:

- **Class Loader.** To load code from other sources, the Java runtime system calls a subclass of the `ClassLoader` (an abstract class in Java), which defines an interface for the runtime system to ask a Java program to provide a class [12]. As a result, we implemented a specialized version of the `ClassLoader` as the skeleton of the agent thread, which can load Java class files from a variety of resources and convert the raw data of a class into an internal data structure representing that class.
- **Agent Context.** We defined and implemented an interface called agent Context, which is responsible for providing an environment in which the downloaded codes executes and interacts with broker.
- **Security Manager.** The capability of downloadable code is powerful, but it is also a potential security threat to a system and raises many concerns. The essence of the problem is that running programs on a computer typically needs to access certain resources on the host. However, if downloaded code is not careful to restrict the access of some critical system resources, it can also provide a malicious code with the same ability to do mischief on the host. In Java, the `SecurityManager` class is meant to define an interface for access control [13,14]. The `SecurityManager` class itself is not intended to be used directly, instead it is intended to be subclassed and installed as the system security manager. The Java platform is designed in such a way that all system calls made by a Java program must be routed through a security manager, which can decide whether or not certain sensitive operations should be allowed. Consequently, we defined a security policy and implemented a specific security manager to support runtime security on host environment. For example, the downloaded code can only access the given directory and file on the local file system.

Monitor Thread

The primary role of the monitor thread is to monitor the status of the managed node. Periodically it will wake up and initiate a request to web server running on the managed node. For minimizing the additional workload added on the managed node, such a request is designed for retrieving a small file (e.g., Home page in our implementation)

If the server responds normally, then the broker will send a message to distributor. Such a message will be treated as "heartbeat" of this managed node. The distributor keeps a counter for each server node, and such a counter will be incremented periodically. When the distributor receives a heartbeat from one server node, it resets the counter associated with that server node. On selecting a server for a new arriving request, distributor checks the counter associated with the candidate server, which is selected by load balancing module. If the counter exceeds a "warning value," which means that the server node may be either unreachable or overburdened. Such a node will be skipped, and the request will automatically be allocated to the next most available server. If the counter exceeds a "dead value" (which is a higher threshold than warning value), the node will be declared dead and be removed from the server cluster. The administration system will automatically record such an event in a log file and inform administrator by E-mail. As a result, any failure in the clustered server can be masked by such a mechanism, and the user from the outside world will be unaware of it.

In addition, the monitor thread also measures the response time that the request took from start to finish and then performs the following algorithm:

```

If (RPT_NOW > RPT_LAST) then
    Interval_time = RPT_NOW * Multiplier;
Else
    Interval_time = RPT_LAST * Multiplier;
RPT_LAST = RPT_NOW;
Sleep(Interval_time);

```

RTP_NOW denote the response time of the request issued by this "wake-up." We also keep the response time of request issued by the last "wake-up" in RTP_Last. Multiplier is a pre-assigned value for calculating the Interval_time. The Interval_time is the interval from this "wake-up" to next "wake-up." There are two main purposes in such a design. First, it will prevent the monitor thread from burdening the load of web server. If the server is overloaded, the monitor thread will decrease the frequency of probe by discovering the longer response time. Second, fine-grained load balancing can be achieved by such a design. That is, if the managed server node is overloaded, the time of interval will be lengthened. This means that the broker will increase the time of interval between this heartbeat and the next. As a result, the distributor will stop to dispatch new request to this node, because it does not receive the heartbeat for a long time.

Agent

The primary role of agent is to perform one delegated task on all nodes according to the request of administrator. An agent is dispatched by controller and executes within the environment created by broker. The agent is in the form of Java classes and is transported across the network as byte streams. It will be reconstituted into Class objects by class loader and

runs in its own thread of execution after arriving at a host.

We first defined an Agent class (a subclass of Object) to be the abstract base class, and then we implemented (inherit the Agent class) all management functions in the form of agent. For instance, we implement an agent to visit all server nodes for updating (or deleting) a file.

In addition, we built in a priority mechanism in the agent system. Based on this, we can assign different priority to different agent. For example, we implemented an agent to analyze the log file of each server node for security concern. Such an agent needs to be executed continually in the system, but it does not need to be executed immediately. Consequently, we can assign a lower priority to this agent. When controller receives a request for executing such agent, it will not dispatch the agent to a server node until that node is idle. Such a design will prevent those background jobs (i.e., housekeeping job) from burdening the load of server node during the high-traffic periods.

Remote Console

For simplifying the system administration as much as possible, we implement a management tool called remote console from which an administrator can issue management operations. The remote console is a Graphical User Interface (GUI) in the form of Java applet, thus it can run under any Java-enabled browser environment. The remote console applet is built completely on top of Java abstract window toolkit (AWT). It will interact with the user and communicate with the controller over the network. At any given time, the Web site manager can download the remote console applet anywhere after proper authentication and then control or monitor the whole system. The applet has a main thread for listening request from the user. Upon receiving a request, it will create a new working thread, and then the main thread is released and ready for other user request. The working thread will talk to the controller for carrying out the requests. We implemented a prototype remote console, which provides the following functions:

- **Configuration function** (e.g., join/remove a server node, or schedules a down time for maintenance): When one select such function, the virtual console will inform the controller, which will invoke the administration function described above to accomplish the configuration setup.
- **Content management** (e.g., add/delete or modify a file): When one selects such a function and fills in the related variables, the remote console will inform the controller, which will dispatch the respective agent to each server node for performing the job.
- **Monitor the activities of the system**: One can select the monitor function to monitor the activities of each node. The controller will invoke

the "read related data" function defined in the control interface to report the related data. The administrator also can assign some special event auditing, and then the controller will dispatch an agent to each node for analyzing its log file.

These user selectable functions are available in the form of buttons at the bottom of the applet's main window. One can see the remote console and its demo operation from our web site [15].

Discussion

In this section, we discuss some related issues raised in our system and compare our system with other related works.

Security

The security of our system depends fundamentally on the following three layers: the Java language itself, class loader, and the security manager. First, the Java language has the following important features from a security standpoint: access control for variables and methods within classes, lack of pointers as a language data type, and automatic garbage collection. Second, the class loader performs further security check to verify that the downloaded code does not violate the security requirements of the Java language. Finally, the security manager supports runtime security on host environment.

Recently, the security manager has been augmented with fine-grained access control mechanisms that allow it to make decisions based on who signed the downloaded code and where it was loaded from [16]. In the future, we will use cryptographic techniques (e.g., digital signature) to guarantee the integrity of code transferred and to provide an identification of the agent provider.

Load balancing

Given a clustered server, poor performance may still exist, which is often due to uneven load distribution throughout the system. As a result, a good load balancing mechanism is required for allocating incoming requests in a way that utilizes the cluster resource evenly and efficiently. At first, we [4,5] used a round robin method as the load-balancing policy for the reason of minimizing the cost associated with load balancing mechanism. This simple scheduling mechanism suffers from the fact that the processing time of individual request is not constant. For instance, the load imbalance still happens while one server node receives five requests for 3KB HTML files, another same server node receives five requests for 3MB MPEG files. Consequently, we implemented an alternative scheduling mechanism to solve this problem. The new mechanism keeps track of the number of outstanding requests in each server node, and distributes the request to the node with the least number of outstanding requests. However, the both two mechanisms do not always reflect the actual load because they do

not track the actual load condition on the server nodes. A potentially better approach is to combine the two mechanisms with some additional information about the actual load. With the administration system, such a multi-level load-balancing can be achieved by monitor thread of broker. The distributor can dispatch the incoming requests to one of the server nodes, using the previous scheduling mechanism, unless the periodic heartbeat of some of these nodes stops. The periodic heartbeat is stopped when the monitor thread detects the fact that the load of the managed server exceeds a critical threshold. In such a case, the distributor temporarily excludes the overloaded servers from scheduling consideration until it receives the heartbeat again (when the load returns under the given threshold).

Comparison

The distributed server concept has been used by many research projects to address the scalability and availability problems of Internet server. In this section, we compare these works with our system (i.e., the distributed server coupled with Java-based administration system).

The NCSA scalable HTTP server is described in [17,18]. Their architecture consists of a number of server machines cooperating to provide the Web service, that use the Andrew file system for sharing content provided by the Web site, and using the round robin DNS server for distributing accesses. In contrast to our scheme, this architecture has the following problems. First, the round robin DNS approach will inevitably suffer from the problems of DNS caching effect [4,5]. In comparison, the load balancing achieved using our routing TCP connection approach is significantly better than that achieved using the round robin DNS techniques. Second, it simply distributes incoming requests in a round robin fashion, which does not consider the heterogeneity of each request and existing load of respective machine. Furthermore, if the configurations (e.g., CPU type, memory size, etc) of machines in the clustered server are different, it is not considered in making a mapping. Third, this architecture did not consider the problem of failure detection and handling.

The Magic Router [19] provides transparent access by placing a modified router before a set of machines, which cooperate to provide a service. Their approach is very similar to our distributor, which redirects incoming Web requests via rerouting TCP connection. In the modified router, they use a user level process to intercept all IP packets and possibly modify packets destined for WWW service for directing it to a selected host. It allocates requests using round robin, random, or incremental system load methods. The main problem of this scheme is that using user space approach will pay the performance cost of context switching delay, and that multiple competing processes will increase the scheduling delay by over an

order of magnitude. Furthermore, it does not consider the content sharing and management problem.

IBM proposed a prototype scalable and highly web server [20] built on an IBM SP-2 [21] system. The system architecture consists of a set of logical front-end nodes and a set of back-end nodes. The front-end nodes run the web daemons and are connected to the external network. The back-end node function as the server node for the sharing file system, used by the front-end to access the data [22]. They use a TCP router approach to dispatch incoming requests. This scheme requires dedicated hardware (i.e., SP-2 system) and software. Consequently, it may not be feasible for all Web sites. In contrast, our approach can be built from commodity hardware and software components. It also can be applied to any existing web site in a manner that any kind of software/hardware of the original server does not need to be replaced.

In addition, several products [23,24,25] have been announced for use as front-end nodes that perform distributing functions across a group of servers. Due to space limitations, we do not describe the details of all these products.

All these works do not address the system administration problem. Many administration or content management operations must be done manually on each node. It should be the nightmare of web site manager, in particular, a number of web site cannot afford specialized computer operations staff. In contrast, we think our system has advantages over other architectures in terms of scalability, high availability, fine-grained load balancing, and powerful and sophisticated system administration functions.

Conclusion

In this paper, we demonstrate the design and implementation of an administration system for distributed server. We exploit the advantages of Java to construct this administration system, which provides the solution for the automation of many tasks in system administration and content management that usually must be done manually. We also offer an easy-to-use GUI for web site manager to maintain and manage the system. With the proposed system, the administrator can perform functions on all nodes at once, monitor the activities of Web site, and manage the entire system as easy as facing a single host. Furthermore, with the feature of downloaded agent, the proposed system provides unlimited possibility to extend its function.

In addition, the proposed system enables multi-level control of incoming requests. The combination of load-balancing mechanism provided by distributor and fine-grained load balancing archived by the administration system gives more precise control for directing incoming requests.

The system also provides excellent high availability features, including automatically detection of failures, and alerts in the form of log file and e-mail. The end user will be unaware that any failure has occurred on the server, although the aggregate capacity of the server will temporarily be reduced. In addition, our approach can enable a high performance server to be built from commodity hardware and software components. Otherwise, it also can be applied to any existing web site in a manner that any kind of software/hardware of the original server does not need to be replaced.

In the future, we will further investigate the security issues raised by Java language and our system in detail.

Acknowledgements

This work was supported by the National Science Council, R.O.C., under contract no. NSC 86-2213-E-110-026 and NSC 86-2213-E-110-032.

The authors would like to thank David D. H. Lin of IBM for his valuable suggestions and help on this work. The authors also greatly appreciate Gretchen Phillips for her proof reading and comments on this paper.

Author Information

C. S. Yang received the B.S. degree in engineering science and the M.S. and Ph.D. degrees in electrical engineering from National Cheng Kung University, Tainan, Taiwan, Republic of China, in 1976, 1984, and 1987, respectively. During 1988-1992, he was with the Department of Electrical Engineering, National Sun Yat-Sen University, Kaohsiung, Taiwan, Republic of China. In 1992, he joined the faculty of the Institute of Computer and Information Engineering at National Sun Yat-Sen University, where he is a professor and Chairman of the Institute now. His current research interests include mobile computing, parallel/distributed programming support environment, and scalable architecture. Reach him at csyang@cie.nsysu.edu.tw.

M. Y. Luo received the B.S. degree in Physics from the National Sun Yat-Sen University, Kaohsiung, Taiwan, Republic of China, in 1995, and the M.S. degree in Computer Science from the Institute of Computer and Information Engineering at National Sun Yat-Sen University in 1997. He has been working toward his Ph.D. degree in the Institute of Computer and Information Engineering at National Sun Yat-Sen University. His research interests are in the areas of computer network, Internet technology, and parallel and distributed system. Reach him at myluo@cie.nsysu.edu.tw.

References

- [1] T. Berners-Lee, R. Cailliau, A. Luotonen, H. Nielsen, A. Secret. "The World-Wide Web" *Communications of the ACM*, Aug. 1994.
- [2] *Java-Programming for the Internet*. <http://java.sun.com>.
- [3] C. S. Yang, M. Y. Luo, "Design an Environment for Scalable Web Server," *Proceedings of 1996 Multimedia Technology and Applications Workshop*, pp. 107-114. Dec. 1996.
- [4] M. Y. Luo, *Design and Implementation of a Scalable and Highly Available Web Server*. M.Sc. Thesis, Institute of Computer and Information Engineering, National Sun Yat-Sen University, June 1997.
- [5] C. S. Yang, M. Y. Luo "Design and Implementation of a Environment for Building Scalable and Highly Available Web Server." *Proceedings of 1998 International Symposium on Internet Technology*, pp. 124-131, April 29-May 1, 1998.
- [6] G. Wright and W. R. Stevens *TCP/IP Illustrated, Volume1*, Addison-Wesley, Reading, May 1994.
- [7] G. Wright and W. R. Stevens *TCP/IP Illustrated, Volume2*, Addison-Wesley, Reading, May 1995.
- [8] T. Berners-Lee, R. Fielding, and H. Frystyk. *Hypertext Transfer Protocol - HTTP/1.0*, <http://www.w3.org/hypertext/WWW/Protocols/>.
- [9] T. Berners-Lee, R. Fielding, H. Frystyk, J. Gettys, J. C. Mogul, *Hypertext Transfer Protocol - HTTP/1.1*, <http://www.w3.org/hypertext/WWW/Protocols/>.
- [10] J. Rodely, *Writing Java Applets*, The Coriolis Group, Inc.
- [11] J. Gosling and H. McGilton. *The Java Language Environment, A White Paper*, May 1996. Available via <ftp://ftp.javasoft.com/docs/papers/langenvron-ps.zip>.
- [12] T. Lindholm and F. Yellin, "The Java Virtual Machine Specification." Addison-Wesley, 1996.
- [13] Joseph A. Bank, "Java Security," Available via <http://swiss-ftp.ai.mit.edu/~jbank/javapaper.ps>
- [14] F. Yellin. "Low level security in Java." In *Fourth International World Wide Web Conference*, Boston, MA, Dec. 1995. Available via <http://www.w3.org/pub/Conferences/WWW4/Papers/197/40.html>.
- [15] <http://pds1.cie.nsysu.edu.tw/WebScale/Demo/console.html>.
- [16] Li Gong and Roland Schemers. "Implementing protection domains in the Java Development Kit 1.2." In *The Internet Society Symposium on Network and Distributed System Security*, San Diego, California, March 1998.
- [17] E. D. Katz, M. Butler, and R. McGrath. "A Scalable HTTP Server: The NCSA Prototype," *Proceedings of First International WWW Conference*. May 1994.
- [18] R. McGrath T. Kwan and D.Reed. "NCSA's World Wide Web Server: Design and Performance." *IEEE Computer*, November 1995.
- [19] E. Anderson, D. Patterson, and E. Brewer. *The Magicrouter, an Application of Fast Packet Interposing*, <http://HTTP.CS.Berkeley.EDU/~eanders/projects/magicrouter/osdi96-mr-submission.ps>.
- [20] D. Dias, W. Kish, R. Mukherjee, and R. Tewari, "A Scalable and Highly Available Web Server," *COMPCON 1996*, pp.85-92, 1996.
- [21] T. Agerwala, J. Martin, J. Mirza, D. Sadler and M. Snir, "Sp2 system Architecture," *IBM System Journal*, Vol. 34, no. 2, pp. 152-184, 1995.
- [22] C. R. Attanasio, M. Butrico, C. A. Polyzois, S. E. Smith, and J. L. Peterson. "Design and Implementation of a Recoverable Virtual Shared Disk." *IBM Research report RC 19843*, T.J Watson Research Center, Yorktown Heights, New York, 1994.
- [23] IBM Corporation. *The IBM Interactive Network Dispatcher*, 1998. <http://www.ics.raleigh.ibm.com/netdispatch>.
- [24] "Cisco LocalDirector," <http://www.cisco.com/warp/public/751/lodir/index.html>.
- [25] Cisco System. *Scaling the Internet Web Servers: A white paper*. http://www.cisco.com/warp/public/751/lodir/scale_wp.htm, 1997.

MRTG – The Multi Router Traffic Grapher

Tobias Oetiker – Swiss Federal Institute of Technology, Zurich

ABSTRACT

This paper describes the history and operation of the current version of MRTG as well as the Round Robin Database Tool. The Round Robin Database Tool is a program which logs and visualizes numerical data in an efficient manner. The RRD Tool is a key component of the next major release of the Multi Router Traffic Grapher (MRTG). It is already fully implemented and working. Because of the massive performance gain possible with RRD Tool some sites have already started to use RRD Tool in production.

Motivation

In Summer 1994, the De Montfort University in Leicester, UK, had one 64 kBit Internet link for more than 1000 networked computers. As it was not possible to get a faster Internet link for another year, it was desirable to at least provide the users on campus with current and detailed information about the status of the link.

This situation prompted the development of the Multi Router Traffic Grapher. Every five minutes, it queried the "Octet Counters" of the university's Internet gateway router. From this data, the average transfer rate of the Internet link was derived for every five-minute interval and a web page was generated with four graphs showing the transfer rates for the last day, week, month, and year. The visual presentation on the Web allowed everyone with a web browser to monitor the status of the link. Figure 1 (next page) presents an MRTG-generated web page.

While the availability of these graphs did of course not increase the capacity of the link, the performance data provided by MRTG proved to be a key argument to convince management that a faster Internet link was indeed needed.

How MRTG-2 Works

The original MRTG program was a Perl script which used external utilities to do SNMP queries and to create GIF images for display on the HTML pages. When MRTG was published on the Internet in spring 1995, it spread quite quickly and people started using it at their own sites.

Soon, however, user feedback highlighted two key problem areas: scalability and portability. While MRTG worked fine when monitoring 10 links, larger sites ran into performance problems. At De Montfort, the intention was to monitor the off-site Internet link and maybe one or two links between buildings; performance was not a limiting factor. External users pointed out that some sites had much larger monitoring needs and were running MRTG right at its limits.

MRTG logged its data to an ASCII file, rewriting it every five minutes, constantly consolidating it, so that the logfile would not grow over time. The logfile did only store slightly more data than was needed to draw the graphs on the web page. The graphs were converted to GIF format by piping a graph in PNM format to the `pnmtogif` tool from the PBM package. This setup limited MRTG to monitor about 20 router ports from a workstation.

A second obstacle for potential users was that MRTG required `snmpget` from the CMU SNMP package. This package proved to be rather difficult to compile on various platforms at that time.

In the mean time, I had left De Montfort University and was working at the Swiss Federal Institute of Technology. There I had no responsibility for the campus network and the Internet link was sufficiently fast. MRTG was not one of my top priority projects anymore. Because the CMU SNMP library did not compile on Solaris I had not even a working installation of MRTG.

This all changed when Dave Rand <daver@bungi.com> got interested in MRTG and contributed a small C program called `rateup`. `Rateup` solved MRTG's performance problem by implementing the two most CPU intensive tasks in C and thus moving them out of the MRTG Perl script. `Rateup` did the logfile rewriting and the graph generation.

`Rateup` initiated the development of MRTG-2.x. First, I modified `rateup` to use Thomas Boutell's GD library [5] which enabled it to generate GIF files much faster than `pnmtogif`. Second, the SNMP portability problem was solved by switching from CMU's `snmpget` to Simon Leinen's Perl SNMP module [4], written in pure Perl and thus making it virtually platform independent.

After almost a year of beta testing and the implementation of many user requested features, the result of these efforts was released as MRTG-2.0 in January 1997.

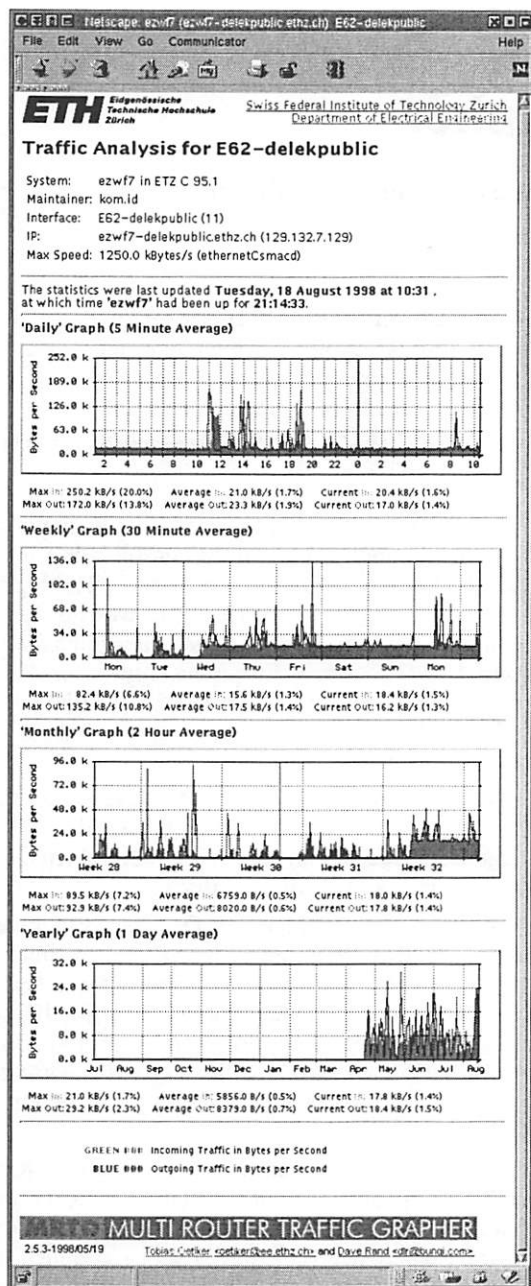


Figure 1: Screenshot of an MRTG-2 web page.

MRTG-2 was not only faster than the previous release, it was also more user friendly. A tool called `cfgmaker`, which is included in the MRTG distribution, is able to build a skeleton configuration file for a router by reading its interface table via SNMP. This allows a lot of people to successfully configure MRTG even when they do not know too much about SNMP or about how to find out which physical router interface is mapped to which SNMP variable.

MRTG-2 does neither require the PBM package nor an external SNMP gatherer anymore. This made the porting of the package very simple. Without using `autoconf` or any similar system, the package

compiles on most Unix platforms. Even a port to Windows NT required only a few changes to the pathname handling and the calling of external programs. The most amazing thing with the NT port was that Simon Leinen's SNMP Perl module worked under NT without change.

MRTG-2 turned out to have the right mix of features to attract the interest of a substantial number of people.

Lossy Data Storage

A key feature of MRTG-2 is its method for maintaining logfiles. The basic assumption for designing the MRTG-2 logfile was that the interest in detailed information about the load of the network diminishes proportionally to the amount of time which has passed between the collection of the information and its analysis. This led to the implementation of a logfile which stores traffic data with a decreasing resolution into the past. Data older than two years is dropped from the logfile. The resolution of the logfile matches the resolution of the graphs shown on the web page. This lossy logfile has the advantage that it does not grow over time and therefore allows unattended operation of the system for extended periods of time. Drawing the individual graphs is relatively fast because no data reduction step is required and thus disk I/O is minimized.

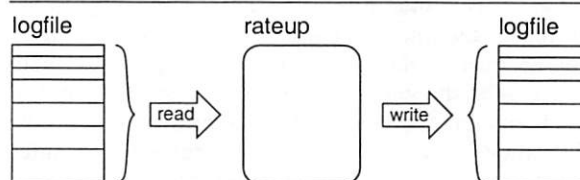


Figure 2: ASCII Logfile processing.

MRTG-2 logfiles are stored in plain ASCII. Each line starts with a time stamp followed by the corresponding traffic data. The file starts with the most current entry and ends about two years in the past. For processing, it is read as a whole, processed in memory and written back to disk. This happens for every single update as shown in Figure 2. Figure 3 shows how the values from the logfile are consolidated over time.

Estimating the Size of the User Base

While it is easy to state that a substantial number of people are interested in a package, it is much more difficult to estimate how many people really use it. Today, the MRTG home page gets about 700 hits and 200 downloads a day. These numbers are quite high for such a page, but it does not show how many sites are actually using MRTG.

Applications like MRTG, which produce output visible on the Web, offer a unique way to measure the size of their user base through the existence of the referrer header in HTTP requests. Every web page

generated with MRTG contains a link to the MRTG home page. Whenever someone comes to the MRTG home page through this link, the web server of the MRTG home page logs the referrer header of the request. With this information it is possible to make a crude estimate about the number of sites using MRTG. Such an analysis was conducted in August 1998, using data from the last two years. It showed referrer headers from about 17500 different hosts under 11400 second level and 120 top level domains.

This excludes all installations of MRTG where the HTML output has been altered to not show the MRTG back-link as well as those where nobody has ever accessed the link. However, it still gives some sort of lower bound for the number of users.

End of Life for MRTG-2

Because of the better performance, more and more large sites started to use MRTG. They soon hit the performance limit again, which was now at roughly 500 ports queried every five minutes. At the same time people started to use MRTG to monitor “non traffic” data sources, requiring more user control over the generated web pages and graphs.

While MRTG-2 initiated widespread use of the package, it was not fundamentally different from the original MRTG-1 Perl script. MRTG had just evolved to the point where it became useful for a larger community. For the maintainer of the package, on the other hand, this evolution had lead to a system which had

outgrown its initial design. Every further enhancement added unproportionally to the complexity of the software. In November 1997, a complete redesign of MRTG was initiated. While some features would be totally new, old strengths were to be preserved.

User feedback and personal experience showed that the following features are the key elements to the success of MRTG-2:

- **Simple Setup:** The configuration is done through simple ASCII text files. An additional tool helps creating an initial version of the configuration file, tailored to a certain router.
- **Easy Maintenance:** Because the logfiles are automatically consolidated on every run and therefore do not grow in size, the system can work unattended for months without running out of disk space.
- **Friendliness:** The HTML pages created by MRTG are easy to understand and give a good visual representation of the network load, providing a sound basis for decisions about upgrading network links.
- **Integrated Solution:** MRTG performs all the tasks required for traffic monitoring. No external database or SNMP packages are required to make it work.

The main problem areas in MRTG-2 are the following:

- **Performance:** MRTG-2 can not monitor more than about 600 router ports in a 5-minute

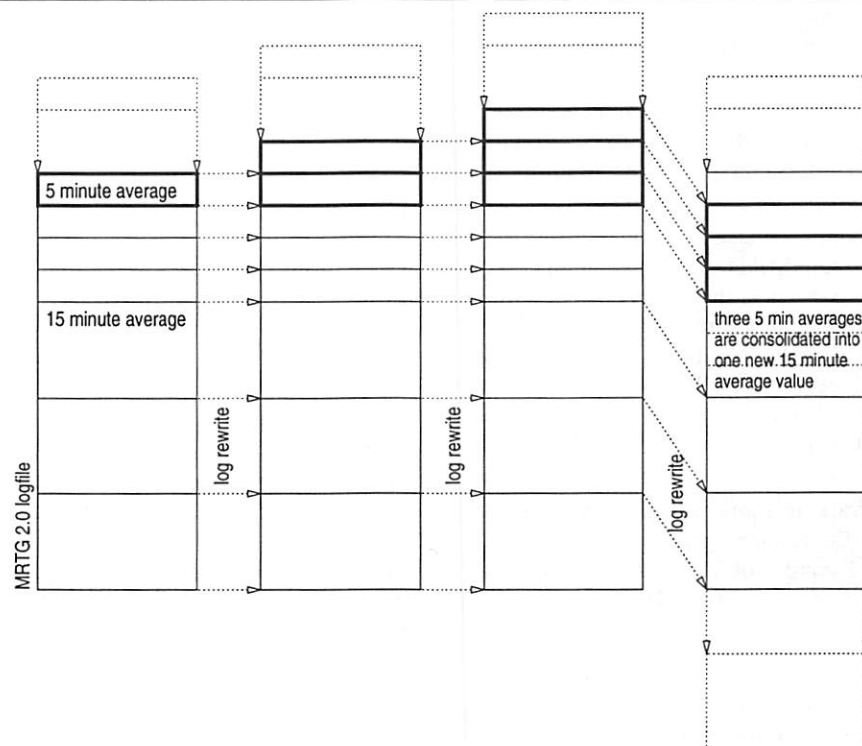


Figure 3: MRTG-2 Logfile processing.

interval, which is due to the way the logfiles are updated as explained above.

- **Flexibility:** While MRTG-2 is quite configurable in general, this seems to make the users especially aware of the areas where configurability is limited, in particular when using the program to monitor time-series data other than network traffic.

The fact that people started using MRTG for tasks it was never designed for, going to great lengths tweaking it to get what they wanted, showed that MRTG offered a unique feature by integrating data collection, storage, consolidation and visualization in a single package. The goals for MRTG-3 were therefore set to be *flexibility* and *speed*.

Design and Implementation of MRTG-3

MRTG-3 moves away from being an application for monitoring network traffic only. The new MRTG will be a toolkit to build applications which monitor large numbers of diverse time-series data sources using a fast data logging facility. It will be able to create a wide variety of graphs, based on data gathered from one or several sources. A parallel SNMP gatherer will help to increase the efficiency of the SNMP data gathering process. The time-critical parts of MRTG-3 are implemented in C, while the glue of the package remains Perl. This allows the users to tailor the package to their needs without recompiling it.

The Round Robin Database Tool

Development of MRTG-3 started with the implementation of a completely new mechanism for data storage. It is called the Round Robin Database (RRD), which gives a clue on how data is stored. The data handling as well as the generation of graphs is implemented in a C program called `rrdtool`. It can either be called from the command line or through Perl bindings.

The Round Robin Database is so much faster and more configurable than MRTG-2 that a number of people have started to use it in their own custom monitoring applications without waiting for the remaining parts of MRTG-3 to be written. After some discussion on the MRTG developers mailing list it was decided to spin off the `rrdtool` into a new package separate from MRTG-3, as it is a complete and useful product all on its own.

Existing software is more useful than planned features. Therefore the remaining part of this section will focus on the Round Robin Database Tool and touch on the other features of the MRTG-3 package only at the very end.

Database Design

Designing a new file format offered the possibility to include a host of features to make the new logfile not only faster but also much more flexible than the old text-based logfile from MRTG-2.

- The RRD format uses `doubles` for data storage. This gets rid of the integer overflow problems seen when monitoring really fast routers with MRTG-2 and it allows to log small numbers like the load of a machine without scaling.
- The RRD can also store `unknown` data values. Which allows it to distinguish between situations where the data input is zero and those when no new valid data can be obtained.
- Parameters like the number of log entries, the resolution of the log and the number of data sources to log in parallel are configurable.
- The data values in the RRD are stored in native binary format. This makes access to the data more efficient, because no conversions are necessary anymore. A cookie in the header of the RRD is used to test if the RRD is compatible with the architecture it is being read from.

Data storage in an RRD is a multi step process. Figure 4 shows a simplified schematic of the new database design and update procedure.

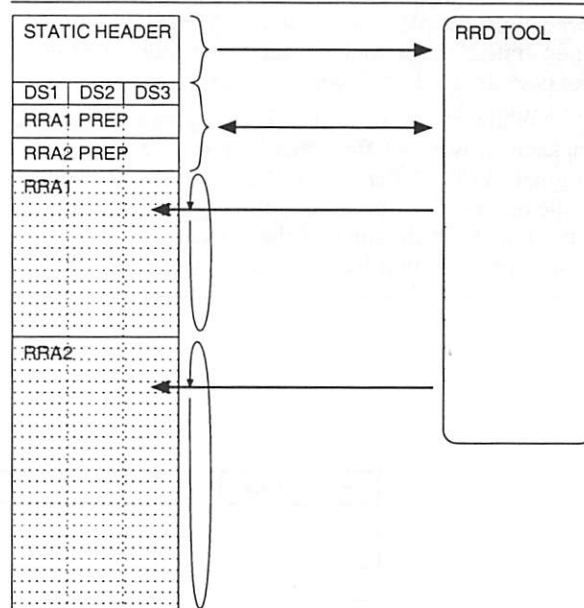


Figure 4: MRTG-3/RRD update procedure.

An RRD can be configured to accept data from a number of data sources in parallel. A data source can be anything, be it an octet counter or the output of a temperature sensor. Each RRD operates at a configurable base time resolution. All data coming from the data sources is re-sampled at this resolution. The re-sampling of the data takes care of the problem that it is not always possible to get new data at the desired point in time but further processing and storage is much simpler when the data is equally spaced along the time axis. Figure 5 shows the re-sampling process for a counter type data source at a 300 second interval. Counter values may arrive at irregular intervals, but data can only be stored in the RRD at fixed interval. The re-sampling ensures that data points are available

for every 300 second interval while the size of the area below the curves is kept constant.

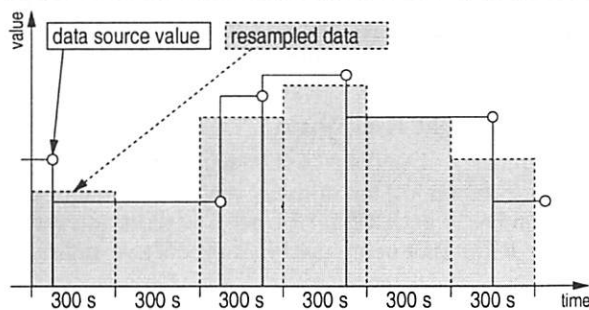


Figure 5: Data resampling process for a counter type data source at a 300 second interval.

The basic idea behind improving logging performance was to reduce the amount of data which has to be transferred between memory and disk. This is achieved by storing data in a round robin manner into preallocated storage areas called Round Robin Archives (RRA) inside the RRD. Each Round Robin Archive has its special properties for time resolution, size and consolidation method. The update interval of an RRA must be a multiple of the base update interval of the RRD. Several values at the RRD's base resolution are consolidated into one value at the RRA's resolution using the consolidation method defined for this RRA. An array of pointers identifies the most current entry in each RRA, such that only one write operation is necessary to update an RRA.

One Round Robin Database (RRD) can contain any number of Round Robin Archives (RRA). For Example, one RRA could be configured to store data at the base resolution of the RRD for a few days, while another one stores the daily averages for 5 years. It is also possible to configure an RRD which mimics the data storage properties of an MRTG-2 log-file.

The time to update an RRD with new data values is roughly proportional to the number of Round Robin Archives it contains plus a constant part for reading the header portion of the RRD and time-aligning new data values.

To help guarantee data quality, the RRD format allows to specify validity conditions like the minimum update frequency required or the minimum and maximum values allowed for a data source. If a condition is not met, the data supplied is regarded invalid and an unknown data value is stored in the RRD.

The new design allows to store in the order of a thousand data values per second in a Round Robin

Database. This rate drops dramatically if the RRD file is accessed via NFS or if the disk cache of the machine is too small compared to the number of RRD files involved in the test. The potential NFS and cache memory problems aside, not much difference was seen between a Pentium 120 running Linux and a Sparc Ultra Enterprise 2 running Solaris at 200 MHz. A direct comparison with MRTG-2 is not possible because MRTG-2 integrates the graph creation into the data logging process.

Graph Generation

MRTG-2 is focused on traffic graphs. Most parameters of these graphs are hard-coded. The graphing engine of the RRD Tool, however, is as flexible as the new RRD format. It allows to produce graphs of any size, spanning an arbitrary time period and to draw data from a number of data sources stored in different RRDs.

Whenever possible, the graphing engine determines sensible default values for its configurable parameters, allowing the user to concentrate on the fine tuning. Almost every aspect of the graph's visual appearance is configurable by overriding the automatic default values. Often configuration is not necessary, because the RRD Tool has several functions which automatically tune features like axis labels and scaling to fit the displayed data.

The graphing part of the RRD Tool also has some built-in analysis capability. It can calculate the maximum, minimum and average values from any data source. For more complex requirements, it is possible to use RPN math on any number of data sources and then graph the result. Figure 6 shows a sample graph demonstrating some of the capabilities of the RRD Tool.

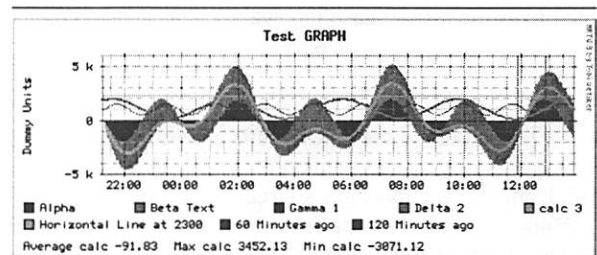


Figure 6: A sample graph from RRD Tool showing some of its features.

Using the RRD Tool

The RRD Tool exists as a stand-alone program called `rrdtool`, which takes its instructions either from the command-line or from a pipe. The preferred way of using RRD Tool, though, is to access its

```
rrdtool create demo.rrd --step=300 DS:COUNTER:400:0:1000000 \
DS:GAUGE:600:-100:100 RRA:AVERAGE:1:1000 RRA:AVERAGE:10:2000 \
RRA:MAX:10:2000
```

Listing 1: Setting up a new Round Robin Database.

functions directly through Perl bindings. This saves the overhead generated from executing a new `rrdtool` process for every operation, and, as opposed to attaching `rrdtool` via a set of pipes to a Perl script, it even works under Windows NT.

Listing 1 shows how to set up a new Round Robin Database called `demo.rrd`.

The `demo.rrd` database has a base update interval of 300 seconds. It accepts input from two data sources and stores its data in two Round Robin Archives. The first data source is a counter type which must be read at least every 400 seconds. It counts between 0 and 1,000,000 units per second. The second data source is a gauge type with a minimum read interval of 600 seconds. It outputs values between \$-100\$ and \$100\$. Any values not complying with the limits defined for each data source will be recorded as *unknown*.

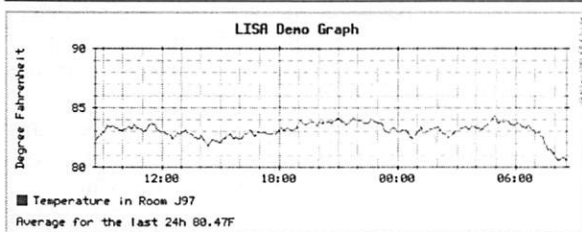


Figure 7: Output of the sample line.

Data is stored in three Round Robin Archives. The first one stores the last 1,000 value sets in 300-second intervals (one base interval). The second RRA stores 2,000 value sets in 3,000-second intervals, building the average of 10 base intervals. Finally the third RRA has the same properties as the second, except that it stores the maximum 300-second values seen over the last 10 base intervals.

Storing data into an RRD is simple:

```
rrdtool update demo.rrd DATA:1994982:U
```

This updates the RRD with a reading from the first data source (the counter) and an unknown reading from the second data source. The time-stamp for this update is the current time. If desired, the time can be specified via the `--time` option.

After the RRD has been filled with some data, the graphing function of RRD Tool can be used to generate a visual representation of the data collected so far. See Listing 2.

```
rrdtool graph demo.gif --start=-86400 --title="LISA Demo Graph" \
--vertical-label='Degree Fahrenheit' \
DEF:celsius=demo.rrd:1:AVERAGE \
"CDEF:fahrenheit=celsius,9,*,5,/,32,+" \
AREA:fahrenheit#ff0000:"Temperature in Room J97" \
GPRINT:fahrenheit:AVERAGE:"Average for the last 24h %2.1fF"
```

Listing 2: Graphing using RRD Tool.

The output created by this command is shown in Figure 7. It shows the temperature data from the last 24 hours (86400 seconds). The temperature data is expressed in degree Fahrenheit and is derived from the Celsius scale using RPN math on the CDEF line in the example above.

RRD Tool in the Real World

Otmar Lendl's (<Lendl@Austria.EU.net>) implementation of an in-house network monitoring solution based on RRD for EUnet Austria is an example for RRD Tool being used in a productive environment.

In Lendl's setup, RRD Tool is used to monitor and graph about 6000 variables from 1200 network interfaces, servers and dial-in lines in 5 minute intervals. Their system runs on a low-end SPARCstation-5/170 at a load of about 0.2. The data-acquisition software is written in Perl 5 using the Perl-bindings for RRD and UCD SNMP. It is split into a scheduler (using the EventServer Perl module) and multiple worker processes which communicate via UDP. The visualization is implemented as `mod_perl` scripts running under Apache 2.3. All images, as well as most of the HTML, are generated dynamically.

Unfortunately, the web pages generated by this is setup are not accessible from outside of EuNet Austria for privacy reasons.

Additional Plans for MRTG-3

SNMP Data Gathering

The Round Robin Database moves the performance bottleneck of MRTG to the data gathering component. The plan for improving SNMP data gathering performance is to issue several SNMP requests in parallel. This works around network latency as well as problems with routers that answer SNMP requests slowly.

Graphs on Demand

Because the generation of graphs is quite expensive, it is not sensible to update thousands of GIF images on a regular basis. It is more efficient to generate the graphs when a user wants to see them. The graph shown in Figure 6 took about 0.3 seconds to generate on a Pentium 120. This means that graphs can be created on the fly and still an acceptable response time can be achieved. For high traffic sites this could be coupled to a graph cache so that the each graph is only regenerated when it is out of date.

HTML Generation

In MRTG-2 the look of the generated HTML pages was tuned using a large number of configuration options. MRTG-3 will work with template files and therefore make the design of HTML pages both simpler and more flexible.

Configuration

While MRTG-2 was a monolithic program, version 3 will be a set of Perl modules which can be assembled into custom monitoring applications. The user can decide which modules to use.

One module will provide a high-level user interface for making MRTG-2-like applications. Scripts which use this module will consist of two parts: In the first part, the user will define all the data sources to monitor. In the second part, an event handler will be called which fetches the requested data and updates the respective RRDs and HTML pages in an optimized order.

Summary

MRTG 1.0 was conceived as an in-house application. Its publication on the Internet showed that there was a considerable demand for such a program. Many free tools like NeTraMet [1], Scotty [6], CMU SNMP were available for retrieving data on the current state of a network link, but MRTG's approach for long term analysis and the friendly presentation on the Web was new. Many users stated that they were able to monitor network links with MRTG "better" than with any commercial tool available at the time.

Today, the NetSCARF project's Scion [7] is providing a solution somewhat similar to MRTG.

BigBrother [3] and CFlowd [2] tools applied in the same area.

The RRD package dramatically improves the logging performance and has better configurability than the MRTG-2 software. This makes it suitable for a broader range of applications, both in the area where a lot of data has to be gathered as well as in cases that call for more complex monitoring configurations than a simple two parameter traffic graph.

Acknowledgments

MRTG is an application which only exists due to the participation of many people. I would like to thank the following people and institutions in particular: Dave Rand for initiating MRTG-2 development with his rateup utility; Simon Leinen for the SNMP library written entirely in Perl; Thomas Boutell for the gd library which is used to generate the graphical output of MRTG; the De Montfort University in Leicester, UK and the Department of Electrical Engineering of the Swiss Federal Institute of Technology in Zurich for supporting the MRTG development; and last but not least to Larry Wall, the creator of Perl.

Availability and Support

MRTG-2 is the current release. It is available from the MRTG home page on <http://ee-staff.ethz.ch/~oetiker/webtools/>. All the code available in connection with MRTG-3 and RRD Tool development is available from <http://ee-staff.ethz.ch/~oetiker/webtools/mrtg/3.0/>. MRTG is available under the terms of the GNU GPL. For exchange with other MRTG users there are mailing lists available. For commercial support please contact the author. Further information can be found on the MRTG home page.

Author Information

Tobias Oetiker got a Master's degree in Electrical Engineering from the Swiss Federal Institute of Technology, Zurich (ETHZ) in 1995. After working for one year at De Montfort University in Leicester, UK doing Unix system management, he returned to Switzerland and has since been employed by the Department of Electrical Engineering of the Swiss Federal Institute of Technology as a toolsmith and system manager. Reach the author at <oetiker@ee.ethz.ch>.

References

- [1] Nevil Brownlee. *NeTraMet, a network traffic accounting meter for PC and UNIX*. <http://www.auckland.ac.nz/net/Accounting/>.
- [2] Daniel W. McRobb and John Hawkinson. *Cflowd, an experimental software to collect data from Cisco's flow-export feature*. <http://engr.ans.net/cflowd/>.
- [3] Sean MacGuire. *Big Brother, a tool for proactive network monitoring*. <http://www.iti.qc.ca/users/sean/bb-dnld/>.
- [4] Simon Leinen. *Perl 5 SNMP Module, an SNMP client implemented entirely in Perl*. <http://www.switch.ch/misc/leinen/snmp/perl/>.
- [5] Thomas Boutell. *GD Lib, a graphics library for fast creation of GIF images*. <http://www.boutell.com/gd/>.
- [6] Jürgen Schönwälder. *Scotty, a Tcl Extensions for Network Management*. <http://www.ibr.cs.tu-bs.de/projects/scotty/>.
- [7] W. Nortong and A. Adams. *Scion, a tool to query SNMP-aware network equipment for performance information, and make that information available on the Web*. <http://www.merit.edu/net-research/netscarf/>.
- [8] Jon Kay. *Internet Measurement Tool Survey*. <http://www.caida.org/Tools/taxonomy.html>.

Wide Area Network Ecology

Jon T. Meek, Edwin S. Eichert, Kim Takayama

– Cyanamid Agricultural Research Center/American Home Products Corporation

ABSTRACT

In an ideal world the need to provide data communications between facilities separated by a large ocean would be filled simply. One would estimate the bandwidth requirement, place an order with a global telecommunications company, then just hook up routers on each end and start using the link. Our experience was considerably more painful, primarily due to three factors: 1) The behavior of some of our applications, 2) problems with various WAN carrier networks, and 3) increasing Internet traffic. "Network Ecology" describes the management of these factors and others that affect network performance.

Introduction

American Home Products Corporation (AHP) is a global life sciences company with over 220 locations. This paper will examine the properties of Frame Relay Wide Area Network (WAN) connections between the Agricultural Research Center in Princeton New Jersey and two European facilities. Then the paper will look at the behavior of network applications on these links.

During the past year AHP started to switch its leased line based WAN to managed Frame Relay networks. Most of the previous WAN usage was for bulk file transfer, database synchronization, light interactive TTY sessions, and some http traffic.

Coincident with the start of Frame Relay implementation, several client-server applications went into testing at the two European sites. These are traditional client-server applications with client PCs in Europe interacting with Oracle databases in Princeton using Oracle's SQL*Net protocol. At the same time, use of the Internet started to increase dramatically. Since the Internet access points for the Corporation are located in the US, this placed an additional load on the WAN.

The old lines did not have the bandwidth to gracefully handle the new demands, so complaints about the performance of the client-server applications were answered with "It should be better with Frame Relay." As the European Agricultural Research sites came onto the Frame Relay network it became obvious that performance did not improve significantly.

We found that initial guesses about the cause of WAN performance problems were often incorrect. With work, they can usually be traced to one or more of the following factors.

- System and Network Administration Practices
- The WAN Carrier's Network
- Commercial Hardware and Software Products
- In-house Application Programs
- Other Uses of the Network

This paper discusses what we learned about managing WAN links, what measurements and monitoring

have helped us, and how we worked with our Frame Relay carriers to improve performance.

Frame Relay Basics

A major advantage of Frame Relay is the ability to burst above the guaranteed bandwidth (committed information rate, or CIR) purchased from a carrier. In the case of the two connections discussed here, CIRs were 64kbps and 32kbps and the access lines varied from 128kbps to 512kbps. Bursting may be limited to multiples of the CIR such as 2x or 4x, or bursting to port speed may be possible. Our Carriers (OC) allow bursting to full port speed, depending on the availability of bandwidth in their core network and the customer's recent usage history. Depending on the policies of the carrier, frames that exceed CIR may be sent with the Discard Eligible (DE) bit set. This allows the carrier to discard those frames if congestion is encountered while they flow through the network. Customers can build credits when usage runs below CIR which may allow bursting above CIR without frames being marked DE. Managing bandwidth use is clearly an important aspect of "Network Ecology."

Network Parameters

In addition to bandwidth, other network performance variables include round-trip-time (RTT) or latency, dropped packet counts, and availability. According to OC packets are dropped only when traffic on a link bursts above the CIR (the DE bit set and the frame encounters congestion). In our experience, availability is very high although regular monitoring is essential. Assuming that bandwidth utilization is under control this leaves RTT as the most important parameter to study.

Minimizing RTT is especially important for interactive TTY sessions and for applications that require a large number of acknowledgment packets. These acknowledgments, sometimes as many as one for each data packet, are due to both TCP and application flow control. In a session involving transfer of many packets, the "wait for acknowledgment" time

adds up quickly. We found that tuning systems and applications so that full-size packets were sent during bulk data transfer portions of a session resulted in the best performance.

On a LAN RTTs are typically <2 ms while trans-Atlantic link RTTs of 90-200 ms are typical. During times of over-utilization, or carrier network problems, RTTs may soar up to eight seconds.

Measuring Bandwidth Usage

It became apparent that we needed to do fairly high-resolution monitoring of network utilization and performance. OC does not normally provide access to the routers that they manage, even those located at customer sites. We were able to negotiate SNMP read-only access which provided several Frame Relay parameters for each PVC (permanent virtual circuit) served by a router.

Every five minutes the following parameters are logged for each PVC: Frames Sent, Frames Received, Bytes Sent, Bytes Received, FECNs, and BECNs. Bytes sent and received are a direct measure of bandwidth usage. The last two parameters, Forward Explicit Congestion Notification and Backward Explicit Congestion Notification are indications of congestion on the network between the end points and may be useful to help detect problems on the carrier's network [Cava98]. The SNMP parameter log is run by cron to ensure that the periods are accurate five minute intervals. The log files are rotated monthly and old logs are retained indefinitely.

Measuring Round-Trip-Time

Since RTT is subject to variation depending on load and routing changes in the OC network, we measure it every five minutes. The RTT measurements double as a connectivity check and are implemented as a mon [Troc97] monitor.

The RTT check monitor sends five small (44 bytes including headers) UDP packets to the echo port of each end-point router. The minimum RTT is used as the reference, but we record the number of packets returned, minimum, mean, and maximum times. If the minimum RTT exceeds a set acceptable limit (currently two seconds), mon alarms are triggered.

If all five of the UDP packets are dropped, then a TCP connection to the echo port is attempted. If the TCP connection attempt times out, the link is considered down and a mon alarm is triggered. About three minutes is required for this process to fail, so we should alarm only on outages that last more than three minutes.

The use of these small probe packets, totaling less than 250 bytes per five minute period, has negligible impact on network capacity.

Communicating Measurement Results

The performance and utilization information collected every five minutes is made available to network managers through Web queries. This allows them to determine if too much bandwidth is being used or if there might be a problem in the carrier's network. Among the parameters supplied on the Web reports is percent of CIR used for both in-bound and out-bound directions. This calculation is based on the five minute average use and, while useful to network managers, is very different from the CIR computed by the Frame Relay switches. The switches use time periods on the order of seconds and compute CIR using algorithms that are not completely known by the carrier's customers.

Other WAN Quality Measurements

In addition to the regular RTT measurements discussed above, we found that measuring RTT vs. packet size is useful. These tests send 1000 random size UDP packets with between 0 and 1472 bytes of random data to the echo port of a router on the other end of a link. All of the results shown here were done at quiet times. The test packet rate was limited by the RTT since we wait (with a 15s timeout) for each packet to return before sending the next packet. The MD5 checksum of the data is computed before the packet is sent and after it is echoed back. This verifies the integrity of the link and eliminates any possible problems with packets that were assumed lost due to the timeout but eventually returned.

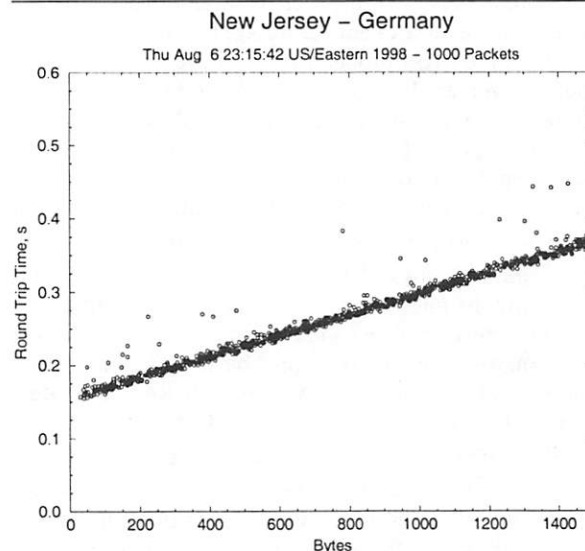


Figure 1a: Round-trip-time vs. UDP packet size. Good performance with 512 kbps and 192 kbps access lines.

By plotting measured RTTs on the y axis and packet sizes on the x axis it is possible to determine the fixed delay (y-intercept), serialization delays (slope), and consistency (scatter of points). The

serialization delay can be predicted quite accurately by just considering the speeds of the access lines on each end of the link (typically 192kbps to 512kbps). The best performing links will have a minimum y-intercept and most points lying close to a straight line. Figure 1 shows three examples of this test on different PVCs.

Table 1 below shows the result of fitting several sets of RTT vs. packet size data. The estimated value was computed using only the speed of the access lines at each end of the link. The measured value includes all serialization delays encountered in the path. The measured fixed delays vary here because the measurements were made over a three month period when the configuration of both our access lines and the core network were changing.

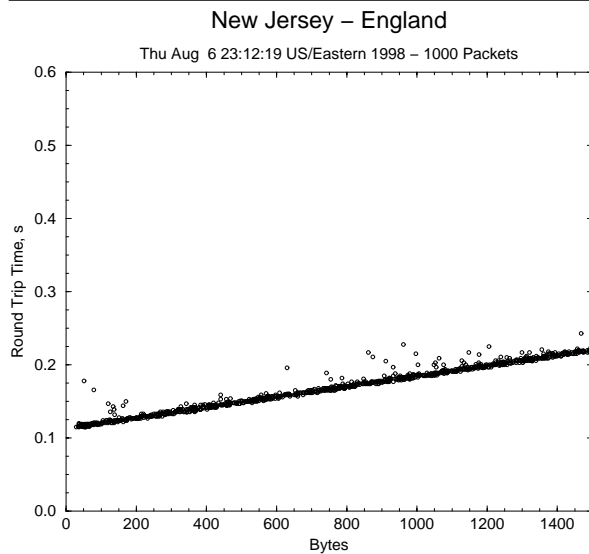


Figure 1b: Round-trip-time vs. UDP packet size. Good performance with 512 kbps access lines on each end.

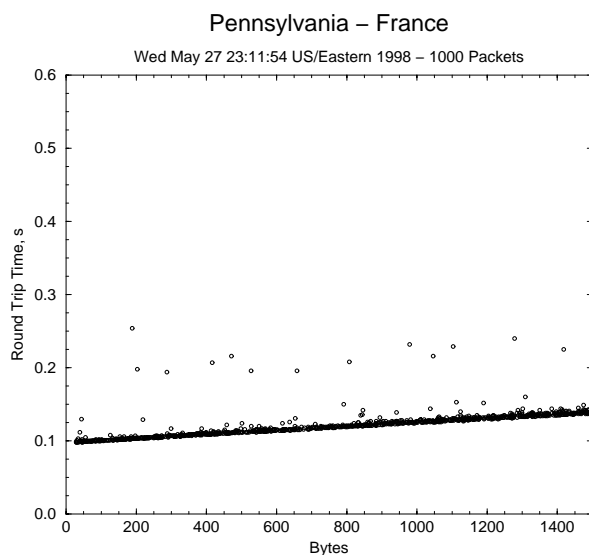


Figure 1c: Round-trip-time vs. UDP packet size. Good performance with T1 and E1 access lines.

Serialization delay improvements can be purchased (up to a point) by paying for faster access lines, while fixed RTT is usually specified only as a target value by WAN carriers and is limited by distance. The table includes measurements made before the New Jersey access line was upgraded from 128 kbps to 512 kbps. The last three table entries correspond to Figures 1a-1c. The difference between measured and estimated serialization delays will include a contribution due to serialization delays in OC's network where there are four additional serialization points per round trip with speeds between 2 and 16 Mbps.

End Points (Port Speed)	Estimated $\mu\text{s/bit}$	Measured $\mu\text{s/bit}$	Measured Fixed RTT, ms
New Jersey (128k) - Germany (192k)	25.4	27.7	150
New Jersey (128k) - England (512k)	19.0	23.1	151
New Jersey (512k) - Germany (192k)	14.0	17.8	155
New Jersey (512k) - England (512k)	7.6	8.9	114
Pennsylvania (T1) - France (E1)	2.3	3.4	99

Table 1: Estimated serialization delays based on access line speeds and measured serialization and fixed delays.

RTT vs. packet size plots can be useful as a measure of service uniformity. Figure 2 shows two plots of measurements taken while OC was experiencing some network instability. The New Jersey - Germany data might indicate route flapping between two, or more, different paths. It is possible that the results of Figure 2a could be due to congestion [Bolo93] either on the PVC, or on the carrier's network. Congestion on the PVC was unlikely in this case since the test packets were essentially the only traffic. Visual inspection of the plots in Figures 1 and 2 suggest that something has changed for the worse in Figure 2. Since the ultimate goal is a largely automatic monitoring system we investigated possible single number metrics that would indicate reduced quality-of-service. The RMS residual (square root of the sum of the squares of the difference between the fit line and the measurements) seems to be a good candidate for this metric. The RMS residuals are 0.306 ms, 0.155 ms, 1.017 ms, and 0.540 ms for Figures 1a, 1b, 2a, and 2b respectively. An OC engineer agreed that Figure 2a indicated a definite problem while Figure 2b was probably within normal operating limits. The best-fit line is shown on each plot.

WAN Quality Measurements - Dropped Packets

Another measure of network quality is the percentage of dropped packets when operating within CIR constraints (below CIR, or bursting with built-up credits). At one point we found that the size of successful ftp transfers from Germany to the US were limited to 25kB, but the reverse path allowed much larger files to be transferred with no problem. Using a custom Perl script that sent numbered UDP packets

we discovered that when packet size went above 966 bytes, every other packet was dropped. We were eventually able to demonstrate this problem using ping with the pre-load option that causes a specified number of packets to be sent as fast as possible.

Unfortunately, many versions of ping, including the Cisco version, do not have the pre-load option. This made it difficult to convince OC's first line support staff that there was a problem. Eventually OC discovered that a Frame Relay buffer size parameter was too small. After they increased the buffer size the problem was corrected.

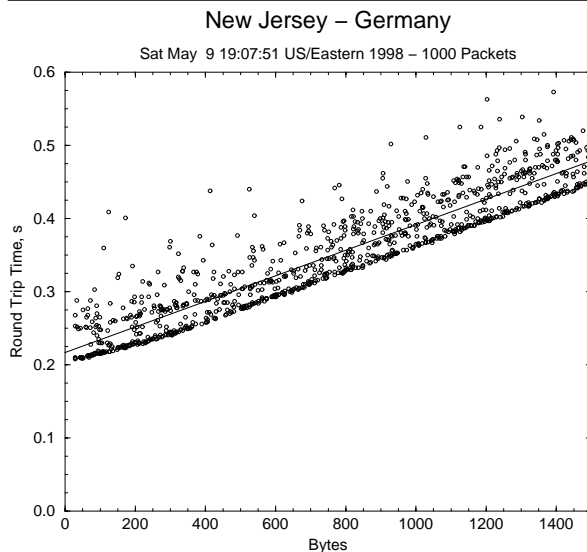


Figure 2a: Round-trip-time vs. UDP packet size, illustrating poor performance.

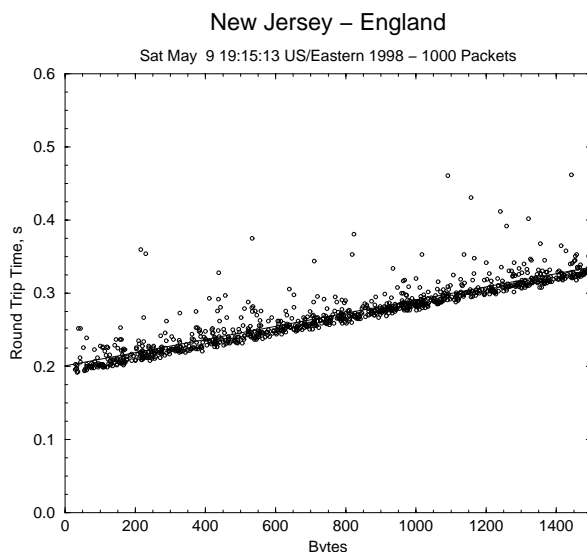


Figure 2b: Round-trip-time vs. UDP packet size, illustrating degraded performance.

Our monitoring program records the number of UDP packets successfully echoed during RTT tests. This provides one measure of the drop rate at a given

time. It would be better to count re-transmitted packets and the number of Frame Relay frames sent and received with the discard eligible bit set. These numbers are not available via SNMP from the routers we are using but could be obtained from another monitoring technique.

What Does the Carrier Monitor?

It took several months before we fully understood what network parameters were pro-actively monitored by OC. It turned out that they only watched for connectivity outages. If their network monitoring system could ping each end of a link, then all was considered well. Furthermore, transient outages were likely to be missed if someone was not watching the network management screen at the right time.

When OC is informed of a customer's negative feelings ("the network seems slow today"), they manually probe deeper to look for problems. Clearly this was not enough; we needed regular measurements of RTTs and bandwidth usage. These measurements are used to establish baselines, trigger alarms when some limit is exceeded, provide reports to assist network management, and build credibility with OC by reporting only real problems.

What's on the Wire? Who's Using the Wire?

After observing that the two European links often had a lot of traffic and that RTTs increased with load, we started to characterize the packets. Using a combination of tcpdump [McCa97], the libpcap Perl module [List97], Network Flight Recorder [Ranu97], firewall logs, and a Network General Sniffer, we were able to determine that some traffic could be eliminated.

There were a lot of routing broadcasts, http traffic to the Internet, and, on one link, Novell broadcast traffic. The Novell traffic was especially interesting since we do not use Novell on either side of that WAN link. It turned out that another Division was using this link to get to their European facilities.

Some of the actual problems discovered via monitoring were:

- For a period of several days, one of the links was fully saturated. Investigation revealed that the Division using our link for Novell traffic was sending large ping packets from three servers in St. Louis to two machines located in Ireland every second. They had been doing some troubleshooting, and forgot to stop the pings. After this had been going on for a week, we brought it to the attention of the responsible network managers and the pings were stopped.
- The printing of purchase orders and other documents from an ERP (Enterprise Resource Planning) system between European sites was very slow. The IT staff responsible for the application suggested that there was "a problem" with

OC's network. By capturing a print session with tcpdump and extracting the PostScript data, it was determined that more than half of the 1.1MB print job was due to trailing spaces that padded every line out to column 140. Another 350kB was due to multiple bitmapped company logos that could be replaced with a 6kB scalable PostScript object. The IT staff is working with the print software vendor to solve these problems.

- Quite a few other high usage problems have been due to "mis-configured" Web browsers, or users leaving their browsers pointing at automatically refreshing pages. The automated firewall reports described below were effective in solving many of these problems.

Where is the Wire?

In our quest to improve WAN performance it seemed that fixed delay was a good parameter to pursue. On the two European links discussed here, fixed delays varied from 150 ms to 225 ms on stable, quiet lines. In contrast, another AHP Division's Pennsylvania to France link had 100 ms fixed delay and the RTT to OC's public Web server located in England, over the Internet was only 90 ms. Therefore, improvement seemed possible.

The fixed delay time for packet transit is due to switching delays and the distance that the signal must travel. We decided to concentrate on distance first.

Several of the US-Europe trans-Atlantic fiber optic cables leave from New Jersey, at least one leaves from Long Island, and some leave from Rhode Island. Our traffic is carried over a number of these cables although we don't know which ones. We were, however, able to learn more about the routes our packets traveled on their way to the trans-Atlantic cables.

Initially our Frame Relay access line was connected to a switch in Maryland, a seemingly round-about way to get to any of the trans-Atlantic cables. When we upgraded the access line speed from 128kbps to 512kbps (to handle capacity requirements and reduce serialization delay) an additional 50 ms was immediately added to the fixed delay. Detective work revealed that because there were no 512k ports in Maryland we were now connected to a switch in Georgia, adding at least 2500 miles to our packet's round-trip. This is especially sad considering that some of the trans-Atlantic cables are located only 35 miles from our Princeton facility.

The signal propagation speed in a fiber optic cable is about 0.66 times the speed of light. This results in a physics limited delay of about 8 μ s/mile. The extra 2500 miles thus represents about 20 ms of fixed RTT. Since the 2500 miles is based on straight

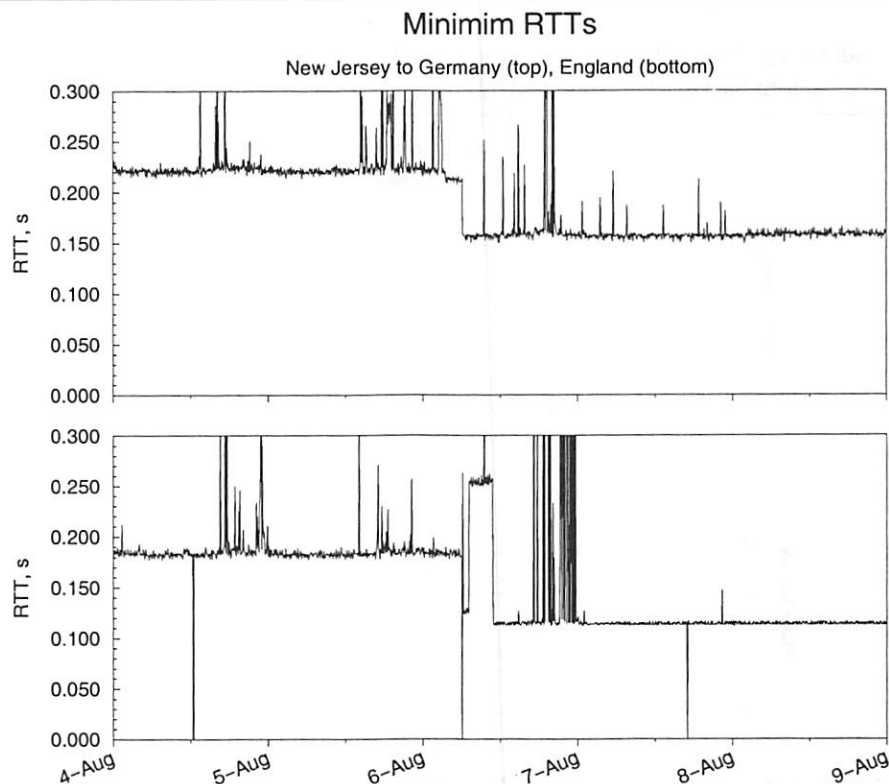


Figure 3: Minimum round-trip-times per five minute interval illustrating improvement due to geographical move of access line.

line distance, and since there must be additional switching delays on this long path, the 50 ms is a reasonable total RTT addition due to the access point change.

After waiting three months we were able to get the access line moved back to Maryland and a 60 ms RTT improvement was immediately realized (10 ms more than we "lost") as shown in Figure 3 (on

6-Aug-1998). While we hoped to get more direct access to the transatlantic cables than passing through Maryland, we were told that the Maryland site is a required stop for Frame Relay packets, unless we wanted to visit Chicago on the way between New Jersey and Europe.

The zero RTT spikes in the New Jersey to England plot indicate short outages. The jump in RTT on

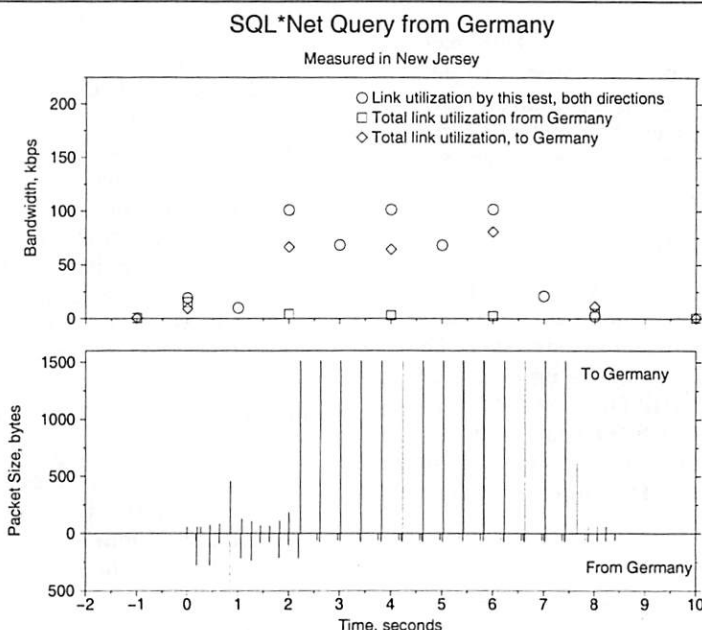


Figure 4a: Bandwidth usage and packet flow for remote database access. Each bulk data transfer cycle consists of about three large back-to-back packets followed by an equal number of acknowledgments.

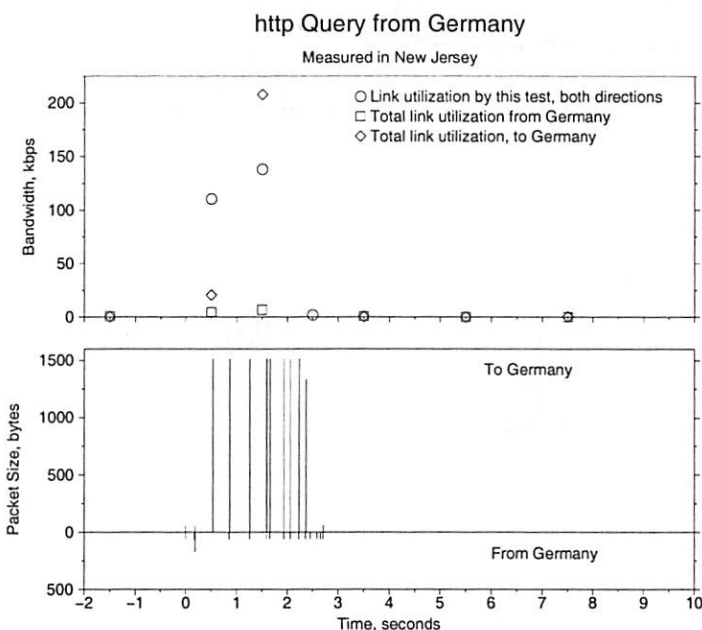


Figure 4b: Bandwidth usage and packet flow for http access of same data as above. The bulk data transfer cycle is similar to Figure 4b except that each set of data packets is followed by a single acknowledgment. Back-to-back packets overlap in both figures.

6-Aug-1998 soon after the access line move was due to an outage in England that caused the probe packets to be routed through Germany. Figure 3 shows quite a few spikes above 300 ms RTT that illustrate how the minimum RTT can increase significantly during times of heavy traffic, usually during working hours.

Analyzing Application Network Usage

In response to complaints about the performance of client-server applications, we captured and analyzed packets for sample sessions. Tcpdump was used for packet capture and our own program for analysis. The test client was a SGI workstation located in Germany. On the client side, tests of SQL*Net were done using a Perl/DBD::Oracle [Bun98] script and the http tests used GNU wget [Nik97]. The servers were Sun SPARC Solaris 2.6 systems running Oracle 7.3.4 and Apache 1.2.6.

The first striking result was that bulk data transfer portions of Oracle/SQL*Net sessions sometimes consisted of many small packets with an acknowledgment for every data packet. On a fast LAN, with sub-millisecond RTTs, this is hardly noticeable; but on a WAN with 100-200 ms RTTs response time quickly adds up to multiple seconds. The Oracle/SQL*Net application was significantly improved by increasing the size of the Oracle row cache on the client side. Figure 4a shows the packet flow in the improved application. Packets in the bulk data transfer portion were mostly full-size, with several sent back-to-back. The client side still sent an acknowledgment for every data packet but several acknowledgment packets were now transmitted back-to-back.

In contrast, performing the same Oracle query using a Web based approach where the SQL*Net traffic stays on the LAN, and only the http traffic passes over the WAN, resulted in improved performance. The http packets were full size without any need for tuning, and up to six packets were transferred before a single acknowledgment was transmitted. The Web based approach was about three times faster than using SQL*Net over the WAN. The Web method transferred about half as much data (due to considerable padding of SQL*Net data). It would, however, not be possible to convert all of the client-server applications to Web technology in the near future. It should also be noted that fancy formatting of the data, such as in a HTML table, would likely result in about the same number of bytes being transferred by both techniques. The SQL*Net vs. http tests are compared in Figure 4. During these tests we monitored the total out-bound bandwidth used on the link (diamonds) and the bandwidth used by the applications under test (circles). Http caused a burst well above the 64k CIR, but finished quickly.

During these tests the time between a burst of data and the associated acknowledgment was usually

between 170 and 350 ms, while the same tests on the LAN gave times between 1 and 8 ms.

References [Stev94] [Stev96] discuss some of the more subtle effects of RTT on network performance such as its effect on TCP window size, timeout, and retransmission, but our simple packet trace analysis made it apparent that RTT was a critical network performance parameter for our client-server applications. We also saw that a significant improvement would result if something could be done on the Oracle/SQL*Net side to enable transmission of more full-size packets.

Setting the Oracle SQL*Net server parameter SDU (Session Data Unit) to 1461 had a much smaller effect than increasing the client's row cache size but resulted in the direct one-to-one mapping of SQL*Net packets to TCP/IP packets. RTT still remains an important parameter that directly impacts performance.

The Role of Internet Traffic

We have found that Internet traffic often consumes a very large portion of the available WAN bandwidth. While there is controversy over the use of Internet usage logs due to privacy and related issues, we have found them to be a very useful tool for managing bandwidth.

At the end of each day we automatically produce a summary of Internet use from firewall logs. The summary includes "Number of Connections and Total Bytes by Network Segment," the "Top 100 Clients" by Number of Connections, Bytes Sent, Bytes Received, and a number of other parameters that do not identify the client's subnet.

The summaries are immediately available via Web pages, and custom reports are e-mailed to network managers with only the information that pertains to the subnets they manage. After being informed of possible problems (by client IP address) through the automated reports, network managers at remote sites have been very successful at reducing unnecessary Internet traffic.

WAN Implementation Suggestions

The following points may be helpful while negotiating with prospective WAN carriers:

- Understand the carrier's network. Get network maps and lists of possible access points.
- Determine what the carrier actually monitors, especially if you are considering a "managed solution." Consider monitoring all important parameters yourself. This will enable your organization to know that they are getting what they pay for, and if they are over utilizing the resource.
- Be sure that you will be able to have read-only SNMP and login access to the carrier's router located on your premises, even if a "managed

solution" is being considered.

- In addition to the usual service level agreement items, such as up-time, repair response, etc., find out what the carrier can specify for RTTs, both minimum and average.
- Make sure that you understand the carrier's problem resolution procedures and how to escalate a problem to a higher level.
- Find out how the carrier notifies customers of system-wide problems. Is there a Web site with network status information?

Future Plans

We expect to develop the ideas presented here further before going into an automatic-only monitoring mode. In particular we want to investigate the following:

- Installation of Web proxy cache servers at many remote sites.
- Implement priority queuing at the routers to lower the priority of packets with destination addresses outside the corporate network (Internet traffic).
- Lowering the priority of packets based on protocol, such as smtp, ftp, and lpd to give interactive traffic the highest priority in router queues.
- Test setting the DE bit for all Internet traffic so that these frames will not count against CIR.
- Comparing the performance of VPNs over the Internet to the private Frame Relay service. We have already made measurements of minimum RTTs to England over the Internet that beat the Frame Relay minimum RTTs by 10 to 40 ms.
- Consider diverting bulk Internet-bound http traffic to Internet access points provided by the WAN carrier.
- Implementing statistical process controls to provide reasonable alarm triggers when a quality-of-service parameter changes significantly. We have already applied this technique to other types of alarms, such as the number of messages waiting in mail queues with good results.

Conclusion

We have discussed a number of techniques, both technical and administrative, that were employed to improve the performance of two trans-Atlantic WAN links. We also described the analysis of application behavior over these relatively low speed network connections, and the impact of several problems that were uncovered by this study.

Among the goals of this work was to keep the two links running smoothly, to develop methods that could be applied to other WAN links in our company, and to determine the ultimate best-case performance of a given link [Bell92]. Knowing the best-case performance, primarily the minimum RTTs, will help choose technology for future client-server applications (i.e., SQL*Net with PC client, other database

protocols with PC client, remote displays on PCs, Web based, or replicated database servers). By tracking the average and worst-case performance we can estimate how often application performance might be unacceptable. Our efforts have already paid off by eliminating the need to install replicate database servers with their high administration costs at the two European locations.

Through the concept of "Network Ecology," which brings together the efforts of system and network administrators, applications programmers, and WAN carriers, we were able to improve the performance of our trans-Atlantic links. An important component of this effort was the development of methods to monitor network characteristics. We intend to continue this work by further automating network and application monitoring tools to keep a close watch over WAN performance with only a small demand on System and Network Administrator time.

Availability

The program for performing connectivity checks and routine RTT measurements (up_rtt.monitor) is part of the mon [Troc97] distribution. The programs to measure RTT as a function of packet size (net_validate) and to read tcpdump output (tcpd_read) may be made available in the future. Readers are directed to MRTG [Oet98] for a system that produces Web based reports on router traffic and other parameters.

Acknowledgments

The authors would like to acknowledge Jim Trocki for many valuable discussions and various pieces of software and Eric Anderson for his detailed review of this paper.

Author Information

Jon Meek is Senior Group Leader of Systems, Networks, and Telecommunications at the American Cyanamid Agricultural Products Research Division of the American Home Products Corp. He received BS and MS Degrees in Physics, and a PhD in Chemical Physics all from Indiana University and has worked in Nuclear and Chemical Physics, Analytical Chemistry, and Information Technology. His research interests include scientific applications of Web technology, systems and network management, data integrity, and laboratory data acquisition. He can be reached at <meekj@pt.cyanamid.com> or <meekj@ieee.org>.

Edwin Eichert is Associate Director of Computer Technologies at the American Cyanamid Agricultural Products Research Division of the American Home Products Corp. Ed received a BS in Electrical Engineering in 1970 and a Masters Degree in Management and Technology in 1991 both from the University of Pennsylvania. His early work, as an Engineer at Westinghouse, was in the design of computer systems to control electric power plants. After Westinghouse he

spent several years doing U.S. Navy sponsored research in holography and electro-sensing in fish. In 1976 returned to the computer industry at Fischer & Porter and FMC. His professional interests include scientific programming and managing technical specialists. He can be reached at <eicherte@pt.cyanamid.com>.

Kim Takayama is Network Manager at the American Cyanamid Agricultural Products Research Division of the American Home Products Corp. He received a BS degree in Microbiology from the University of Maine at Orono and has worked as a Genetic Toxicologist for Exxon Biomedical Sciences, followed by seven years of applications development. He is currently in his seventh year of managing networks and systems for Cyanamid. He can be reached at <takayamak@pt.cyanamid.com>.

References

- [Bell92] Steven M. Bellovin, "A Best-Case Network Performance Model," February 1992. <http://www.research.att.com/~smb/papers/index.html>.
- [Bolo93] Jean-Chrysostome Bolot, "Characterizing End-to-End Packet Delay and Loss in the Internet," *Journal of High Speed Networks*, Volume 2, Number 3, pp 305-323, 1993.
- [Bun98] Tim Bunce, "DBD::Oracle - an Oracle 7 and Oracle 8 interface for Perl 5," available from CPAN mirrors, see <http://www.perl.com>.
- [Cava98] James P. Cavanagh, "Frame Relay Applications: Business and Technology Case Studies," Morgan Kaufmann, 1998.
- [List97] P. Lister, "Net-Pcap-0.01," 1997.
- [Nik97] Hrvoje Niksic, "GNU wget" available from the master GNU archive site prep.ai.mit.edu, and its mirrors.
- [McCa97] Steve McCanne, Craig Leres, Van Jacobson, "TCPDUMP 3.4," Lawrence Berkeley National Laboratory Network Research Group, 1997.
- [Oet98] Tobias Oetiker, "MRTG, Multi Router Traffic Grapher," *12th Systems Administration Conference (LISA)*, 1998.
- [Ranu97] Marcus J. Ranum, Kent Landfield, Mike Stolarchuk, Mark Sienkiewicz, Andrew Lambeth, and Eric Wall. "Implementing a Generalized Tool for Network Monitoring," *11th Systems Administration Conference (LISA)*, 1997.
- [Stev94] R. Stevens, *TCP/IP Illustrated, Volume 1: The Protocols*, Addison-Wesley, 1994.
- [Stev96] R. Stevens, *TCP/IP Illustrated, Volume 3: TCP for Transactions, HTTP, NNTP, and the UNIX Domain Protocols*, Addison-Wesley, 1996.
- [Troc97] Jim Trocki, "mon, a general-purpose resource monitoring system," <http://www.kernel.org/software/mon/>.

Automatically Selecting a Close Mirror Based on Network Topology

Giray Pultar – giray@coubros.com

ABSTRACT

The content of many popular ftp and web sites on the Internet are replicated at other sites, called “mirrors”; typically, to decrease the network load at the original site, to make information available closer to its users for higher availability; and to decrease the bandwidth requirements these sites place on long-haul network connections, such as international and backbone links.

Even though the success of mirroring depends heavily on the selection of a good mirror, there are very few methods to pick a good mirror: i.e., a mirror “close” to its user based on network topology.

This paper describes a method and two tools developed to locate a “close” mirror among replicated copies of a network service such as ftp, www, irc, streaming audio by utilizing network topology information based on autonomous systems. Routing information from the Internet Routing Registry is combined with information about the location of mirrors to generate mirroring tables, similar to routing tables, which are used to identify a “close” mirror, where “close” is defined as traversing the minimum number of autonomous systems.

The tools are available via anonymous ftp from ftp.coubros.com.

Background Material

Mirroring

“Mirroring” is the replication of the content of a site (such as a web site or an ftp site) at a different site called a “mirror.”

The content at the original site is replicated at the mirror, typically once a day, by utilizing a mirroring program, such as `mirror` [6].

In most cases, the locations of the mirrors are advertised at the original site, and users of the site are asked to choose a mirror “close” to themselves. These mirrors are called “public mirrors,” because their existence is advertised at the original site; and users of the original site are allowed to use any of these mirrors. Sites are mirrored at public mirrors to decrease the network and server load at the original site and to make the content available to the public even if the original site is down.

In some cases, the location of a mirror is not advertised at the original site. Such a mirror is called a “private mirror,” because its existence is not advertised at the original site; and the mirror is intended for a subset of the users. Sites are mirrored at private mirrors to decrease the network load at the mirror location (based on the assumption that the content will be requested by the users at the mirror location more than once) and to make the content available to the users of the private mirror even if the original site is down.

In this paper, we are only concerned with public mirrors.

Autonomous Systems

Section 2.2.4 of RFC 1812 [3] gives the definition of an autonomous system:

“An Autonomous System (AS) is a connected segment of a network topology that consists of a collection of subnetworks (with hosts attached) interconnected by a set of routes. The subnetworks and the routers are expected to be under the control of a single operations and maintenance (O&M) organization. Within an AS routers may use one or more interior routing protocols, and sometimes several sets of metrics. An AS is expected to present to other ASes an appearance of a coherent interior routing plan, and a consistent picture of the destinations reachable through the AS. An AS is identified by an Autonomous System number.”

Figure 1 depicts a hypothetical internet made up of three autonomous systems. Autonomous system 1 (AS1) has two routers; AS2 has three and AS3 has two routers. There are external network connections between AS1 and AS2; and between AS2 and AS3. This hypothetical network will be used throughout this paper.

Exterior Gateway Protocols

Section 7.3.1 of RFC 1812 [3] states: “Exterior Gateway Protocols are utilized for inter-Autonomous System routing to exchange reachability information for a set of networks internal to a particular autonomous system to a neighboring autonomous system.”

In our hypothetical world, for example, AS1's router would tell AS2's router that it can reach all the networks inside AS1 (such as the network containing client 1). Similarly AS2's router would tell AS3's router that it can reach all the networks inside AS2, (such as the network containing Server 1). Moreover, this AS2 router, would also tell AS3 that it can reach the networks connected AS1.

With the exchange of this information, AS3 would learn that it can reach Client 1 via AS2. The protocols used for exchanging this information are called External Gateway Protocols.

It is beyond the scope of this document to describe the details of exterior gateway protocols, such as BGP, and the reader is referred to [2].

The Internet Routing Registry – IRR

There are three components to understanding the motivation for the internet routing registry (IRR) [4] : exchange points, policy routing and route servers.

Exchange Points

The Internet is a collection of interconnected networks managed by different network providers. When a host is connected to the internet; it is connected to one of the networks that make up the Internet. The fact that a host on the Internet can reach any other host on the internet is the result of agreements between the network providers and the connections between the different networks that make up the internet.

If each provider had to connect to every other provider on the Internet directly, the number of network connections required would grow with the square of the number of providers; and therefore would be impractical.

The solution to this problem was the creation of exchange points (aka network access points). Each network provider gets connected to an exchange point, and can then communicate with all other providers connected to the same exchange point. In this scenario; the number of network connections required grows linearly with the number of network providers.

There are currently several exchange points in the US, such as MAE-EAST, MAE-WEST, PAIX, etc. For more information about exchange points, the reader is referred to [5].

Policy Routing

As described in the previous section, there are several network providers that are connected to each exchange point. However, this does not mean that each provider is willing to exchange traffic with all other providers at the same exchange.

When a provider does not want to send traffic to another provider, they must modify the routing tables on their routers that connect to the exchange point to direct this traffic to a different provider.

Generating routing tables, not only based on network connectivity, but based on the preferences or policies of a provider is called "policy routing."

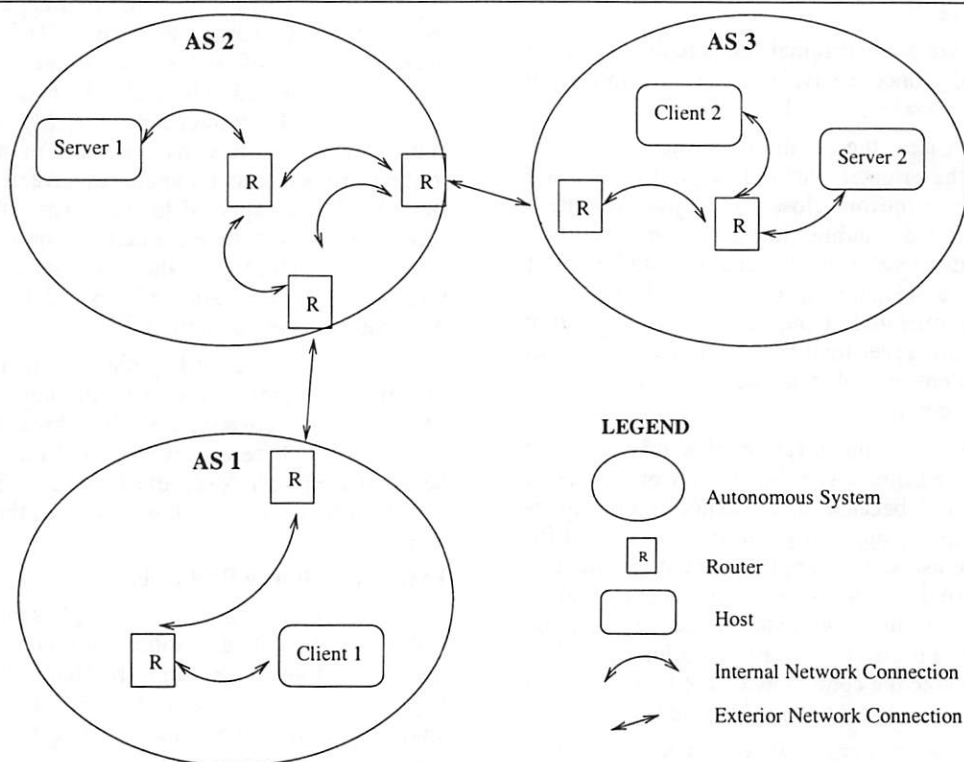


Figure 1: A hypothetical internet with 3 autonomous systems.

Route Servers

There are two main functions performed by a router:

1. routing packets: receiving packets and sending these packets out based on IP addresses and the router's routing tables
2. routing table management: communicating with neighboring routers and updating routing tables and calculating new routes based on changes in network connectivity

"Route servers" are computers (typically UNIX based general purpose computers, as opposed specialized hardware such as routers) designed to offload the routing table management function from the routers, so that they can focus on routing packets. There are typically two or more route servers at each exchange point. These route servers communicate with all of the routers at the exchange point, collect connectivity information and route advertisements, and prepare routing tables for all of the routers based on the policies of each network provider. The routers at the exchange point, communicate with the route server and receive their routing tables specifically prepared for them.

To prepare routing tables for each router at the exchange, the route servers need to know the policies of each Internet Service Provider to honor their preferences. The Internet Routing Registry (IRR) is a collection of routing policies of each ISP to be used by route servers in preparing the routing tables.

The information that makes up the IRR is currently stored in five different databases, provided by five different organizations. For more information on the IRR, see [4].

The databases in the IRR are stored in the RIPE-181 format [8] and contain objects that describe network providers, their contacts, autonomous systems and routes. The objects of interest to this paper are the "route" object, and the "autonomous system" object.

The "route" object defines a route in the CIDR format, and its originating autonomous system.

```
route: 198.87.45.0/24
origin: AS3333
[...]
```

This tells us that the address space 198.87.45.0-198.87.45.255 originates in AS3333.

The "autonomous system" object defines an autonomous system and a list of its neighboring autonomous systems and what routes it is willing to advertise, and what routes it is willing to accept from those autonomous systems. For example, see Listing 1.

This object tells us that autonomous system AS1104 connects to AS1213 and is willing to accept route advertisements from AS1213 for all routes that are registered in the IRR with an origin of AS1213. It is also willing to accept any route from AS1755. In advertising routes, AS1104 will advertise all the routes it knows about to AS1213; and all the routes registered as originating from AS1213 and itself from AS1104. It is likely that this autonomous system connects to the Internet via AS1104, and AS1213 connects to the Internet via this autonomous system. (However, this is not conclusive, as there may be other entries in the IRR that describe how AS1213 is connected.)

Existing Methods

There are relatively few existing methods of selecting a "close" mirror.

- **User selection:** This is the method that is most commonly used. The user is given a list of all the mirrors and is asked to choose one which they think is "close" to them.
- **Geographical:** This method is a slightly improved version of the "user-selection" method. The user is, again, given a list of all the mirrors along with their geographical locations; and is asked to choose one which they think is "geographically close" to them (e.g., www.gnu.org).
- **Connect everywhere:** This method is really not a method for selecting a "close" mirror; but making a site "closer" to its users. The idea is to get connected to many countries and many exchange points, so that the server is "close" to everyone (e.g., www.digisle.net).
- **Domain name:** This is one of the two existing methods found that can automatically select a close mirror. It tries to match as many labels as possible from the fully qualified domain names of the server and the client. For example, a client at joe.domain.com would use ftp.domain.com if there were such a mirror, since the domain.com portion of the

```
aut-num: AS1104
as-in: from AS1213 100 accept AS1213
as-in: from AS1755 150 accept ANY
as-out: to AS1213 announce ANY
as-out: to AS1755 announce AS1104 AS1213
[...]
```

Listing 1: Route acceptance example.

names match (e.g., `www.perl.com/CPAN`).

- **Router based:** The other method to automatically select a close mirror is by using information from routers at the mirror sites. For example, Cisco's Distributed Director product [7] uses Cisco's Director Response Protocol (DRP) to find a close mirror. This method requires that all the routers at the mirror sites implement DRP, and therefore use Cisco IOS.

Initial Ideas

Following are some of the ideas generated in formulating solutions to the problem of "finding a close mirror."

Client-side Selection

One of the first issues looked at was the advantages and disadvantages of finding a "close" mirror at the client side.

The first issue with a client side solution is that the software must be installed at the client to perform the mirror selection. This problem has recently been somewhat resolved by client side scripting and virtual machine technologies, such as Java and Javascript. Loading software on to the limited number of mirrors is easier than loading software onto the clients.

We assume that the server, already contains a list of its public mirrors. (This is probably a valid assumption in most cases on the Internet today.) It would be unreasonable to expect the client to have a priori knowledge of mirrors for all sites. Even if the client could look up a list of mirror sites, this information would most likely be coming from one of the mirrored sites, anyhow.

Implementing a client side solution would make sense if there was any information that clients already possessed which was necessary to make the "closest" decision; but was also very difficult to transmit to the server side. The only piece of information that comes to mind that the server does not already possess is the IP address of the client; however, the transmission of this information is very simple.

Based on the software loading problem and the location of data already available to make a decision, in our opinion, the method of finding a "close" mirror must work on the server side.

Traceroute

One of the tools that comes to mind when talking about route paths on the internet is `traceroute`. What kind of mechanism can we implement using `traceroute`?

One possibility is to have each mirror run a `traceroute` to the client, and among themselves decide which one has the shorter path. There are two problems with this approach:

1. All the mirrors need to be contacted, and work together to make a decision. As the number of mirrors increases, this will take increasingly more traffic/time.
2. This method will give the paths from a server to the client. However, the paths on the internet are not always symmetric. That is, the path taken from a client to a server is not the same as the path taken from the server to the client. Assuming that most of the data transfer is going to be from the server to the client, it would be inappropriate to look at the path in the opposite direction of the bulk of the data transfer.

The other possibility is to run `traceroute` from the client to all the mirrors. Again, as the number of mirrors increases, the time/traffic to make a decision will increase with the number of mirrors.

Based on the scalability issue and asymmetric nature of the internet, would be an inappropriate mechanism; and that the mechanism we find must be scalable: the time/traffic requirement must not grow with the number of mirrors.

The Solution

The method being implemented in the tools presented in this paper is to use the routing information available from the sources that provide autonomous system path information (such as the Internet Routing Register) to determine, a priori, a "close" mirror, that is the mirror reachable by traversing the minimum number of autonomous systems, for all IP addresses by building a mirroring table.

This mirroring table can then be used to make decisions as to which mirror is "close" to a client with a given IP address.

Some of the assumptions and implementation issues relating to this solution are discussed in the "Discussion" section.

Source Address	Mirror address
8.0.0.0/8	http://www.us.domain.com/public/tool
28.10.0.0/16	http://www.domain.de/publik/
130.36.0.0/24	http://www.us.domain.com/public/tool
130.36.128.0/28	http://www.domain.de/publik/

Table 1: Mirroring table.

Mirroring Table

The mirroring table is a table that specifies which mirror clients should use based on their IP address. This table is similar to routing tables: routing tables specify which interface should be used for sending packets and the next hop, based on the destination IP address of packets. A mirroring table specifies which mirror should be used, based on the IP address of the client.

As an example, consider a primary site at `http://www.us.domain.com/pub-lic/tool` (shown as Server 1 in AS 2 Figure 1) and a mirror at `http://www.domain.de/publik/` (shown as Server 2 in AS3 in Figure 1).

Furthermore, let's assume based on the network topology information, we generate a mirroring table, of which a portion looks like Table 1.

The source address is specified using the CIDR syntax [9]. According to this table, client 1 at 8.2.3.4 (in AS 1) would be directed to use the primary site (server 1), based on the first line. On the other hand, Client 2 at 130.36.130.22 (in AS 3) would be directed to use the mirror (Server 2) according to the last line of the table.

Generating the Mirroring Table

The IRR contains information about network routes, and route advertisements that ISP's are willing to send and accept from other ISP's.

One can visualize the information in the routing registry as a graph, by considering the autonomous systems as nodes, and the sending and the willingness accept route advertisements as arcs between these nodes.

Once represented as a graph, the problem of finding a "close" mirror is reduced to applying Dijkstra's one-to-all shortest path algorithm for each mirror to generate AS paths from each AS to the AS containing the closest mirror.

Once we know which mirror each AS should use, we can enumerate all the routes originating from each AS to generate the mirroring table.

Optionally, the mirroring table can be processed through a route aggregation algorithm, to decrease the number of entries in the mirroring table.

Directing the Clients

The ease of directing a client to a "close" mirror for a given protocol depends on whether the protocol supports redirection or not. For example, the http protocol supports the "Location:" header which can redirect its client to any other URL.

A http redirector, would take the IP address of the client, look it up in the mirroring tables, find a "close" mirror, and return the "Location:" header with that mirror.

For protocols that do not support redirection, one possibility is to implement a modified DNS server. The server would be use the IP address of the resolver requesting a name resolution as the client address to find a "close" mirror. For an example of a modified name server, see [10]

Implementation

The mkmirrortable tool

The mkmirrortable tool takes two or more arguments and generates a mirroring table on stdout (in the format shown above). The arguments are:

- the name of a mirror file, in a /etc/hosts format with addresses in the first column, and a label (such as its hostname, or its URL) in the second column and
- name of IRR database(s) file, in the RIPE-181 format [8].

The execution time of mkmirrortable depends on the length of the IRR database files. On the current IRR database, on a Pentium 200 processor, it takes several minutes to generate a mirroring table.

The closest.cgi Tool

The closest.cgi is a simple Common Gateway Interface (CGI) perl script, that redirects a web browser to a mirror. This is accomplished via the "Location:" header returned by the cgi program.

The client IP address is delivered to closest.cgi via an environment variable. The tool then opens the mirroring table generated by the mkmirrortable program and searches for all entries that match the IP address of the client. The route with the longest prefix (the most specific route) is selected and its mirror is returned to the web server via the "Location:" header.

Discussion

There are several assumptions that have been made in order to use autonomous systems to find a "close" mirror and in our implementation.

Definition of "close"

Earlier, we defined a "close" mirror, as the mirror that is reachable by traversing the minimum number of autonomous systems. However, this definition is rather arbitrary (and rather self serving for the purposes of this paper).

There is some validity to this definition: The autonomous system paths can be thought of as a simplification of all routes on the Internet. It would be very difficult to map all the routes in the entire Internet. Even then, using this map and doing calculations based on such a map would be extremely difficult. The autonomous system map is a representative derivative.

Moreover, in general, crossing autonomous systems crosses organizational boundaries, and there are monetary costs in crossing organizational boundaries.

Even though the users are not directly aware of it, there are costs associated with crossing network providers: there are the operational costs of exchange points, as well as charges imposed by larger network providers to use their networks. (Of course there are some exceptions, where network providers agree to exchange traffic with each other for free.)

To improve this definition of "close," one would have to look at the reasons that a site or its users wish to use mirrors.

In most cases, the user does not care which mirror is used to deliver the service, but would like to be connected to a "up" mirror with high bandwidth and/or low latency depending on the service.

The owners of the site, on the other hand, may wish to redirect users to mirrors, to conserve bandwidth at their own site, or to manage the distribution of users among mirrors.

From an Internet architecture perspective, mirrors can serve two purposes: to decrease the overall bandwidth used on the internet, and to avoid some paths becoming overloaded.

A better definition of "close" would need to take into account the purposes of all of these parties.

AS Paths Represent Network Routes

We have assumed that the network route from a client to a mirror will pass through the autonomous system path that is constructed by looking at the autonomous systems that claim to originate the routes for the client and the mirror.

This is most likely true for most routes in the Internet. Almost all network providers represent their network as autonomous systems, and present a coherent view of their network to all other networks at the exchange points. Moreover, since the routing tables at the exchange points are built from the Internet Routing Registry, the coherent view presented by each network provider is stored in the IRR.

An exception to this would be a private network connection between two network providers that does not pass through an exchange point, and is not represented in the IRR.

Different AS sizes

We have made an implicit assumption that all ASes are of the same size, by ignoring the size of ASes. We calculate the number of ASes traversed without regard to how large they may be.

In reality, the size of autonomous systems vary greatly. A better approach would be to try to estimate the size of each AS, and try to minimize the total sum of the sizes of the ASes traversed.

AS Cost of Routes Ignored

One of the pieces of data available in a AS definition in the IRR is a cost associated with routes learned from neighboring autonomous system.

In this implementation, this cost is ignored. It would be difficult to take this into account however, since the cost is only relative to other neighbors for the same AS, but has no significance when going across ASes.

Asymmetric Internet and Direction of Data Flow

Earlier in this paper, we talk about the asymmetric nature of the Internet. We have assumed that the direction of data flow is from a mirror to the client; and have ignored that there is some data flow from the client to the mirror. It is possible to think of services where there is equal or more data flowing from the client to the server (e.g., sending e-mail).

A better method for choosing a "close" mirror would look at the data flow requirements in both directions to try to find an optimal path.

Future Directions

This section discusses several possibilities for extensions to this work.

Empirical Evaluation

An extension to this paper and a well contained project would be to implement the mirror selection method presented in this paper at several sites, and collect data on the selections made. This data can then be analyzed to see how useful this method is and how valid the assumptions being made are.

Client-side Solutions

Another approach would be to look at solutions where the client is involved in the selection of a "close" mirror, either solely at the client, or in cooperation with a server. This could be implemented in a browser downloadable language such as Java or Javascript.

Dynamic Route Information

This implementation uses static information in the Internet Routing Registry. Using dynamic routing information by peering with routers using BGP-4 would make it possible to take into account link states between providers in choosing a mirror.

Modified DNS Server

For services that do not support automatic redirection, using the domain name service as a redirector may be possible. A modified DNS server could respond differently depending on the client. The IP address returned by the nameserver would be the address of a "close" mirror to the client that requested the address. This would work based on the assumption that the resolver requesting the address is in the same autonomous system as the client requesting the service.

Migration to RPSL

There is a plan in the Internet Registry to migrate to a different route specification language called RPSL. The `mkmirrortable` program would need to be extended to use RPSL instead of RIPE-181.

Using Latency and Bandwidth Availability Metrics

A better "mirror" selection, from the client's perspective would be to find a mirror with a low latency and/or the highest possible bandwidth. Even better, would be to be able to reserve the bandwidth from a specific mirror.

Conclusion

This paper describes a method and presents two tools developed to locate a "close" mirror among replicated copies of a network service by utilizing network topology information based on autonomous systems.

The author hopes that this breakthrough will be embraced by most ftp and web service providers, and that he does not have to ever "choose his own" close mirror again :-).

References

- [1] Huitema, Christian *Routing in the Internet*, ISBN 0-13-132192-7, Prentice Hall, 1995 Englewood Cliffs, New Jersey 07632.
- [2] RFC1771 - *A Border Gateway Protocol 4 (BGP-4)*.
- [3] RFC1812 - *Requirements for IP Version 4 Routers*.
- [4] <http://www.merit.edu/radb/docs/irr.html>.
- [5] <http://www.isi.edu/div7/ra/>.
- [6] <http://sunsite.org.uk/packages/mirror/>.
- [7] http://www.cisco.com/warp/public/751/distdir/dd_wp.htm.
- [8] Tony Bates, Elise Gerich, Laurent Joncheray, Jean-Michel Jouanigot, Daniel Karrenberg, Marten Terpstra, Jessica Yu, *Representation of IP Routing Policies in a Routing Registry*, <http://www.merit.edu/radb/docs/ripe-181.html>.
- [9] RFC1519 - *Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy*.
- [10] Schemers, Roland J. III, "lbnamed: A Load Balancing Name Server in Perl," *LISA '95*. <http://www.usenix.org/publications/library/proceedings/lisa95/>.
- [11] RFC2280 - *Routing Policy Specification language*, <ftp://ftp.isi.edu/in-notes/rfc2280.txt>.

What to Do When the Lease Expires: A Moving Experience

*Lloyd Cha, Chris Motta, Syed Babar, and Mukul Agarwal – Advanced Micro Devices, Inc.
Jack Ma and Waseem Shaikh – Taos Mountain, Inc.
Istvan Marko – Volt Services Group*

ABSTRACT

Moving a division of approximately 200 employees from one building to another across town can be a daunting task. It involves coordination among teams from systems administration, networking, facilities, and security as well as support from management and cooperation of the employees being relocated. Contractors and subcontractors are frequently hired to handle physical relocation of goods from one location to another, construction of new server rooms, electrical rewiring, installation of new cooling systems, etc. This paper is the story of how we handled the move and reconfiguration of a network of approximately 1000 nodes over a long weekend in May 1998.

Previously published work has discussed some of the issues that challenged us here. The reconfiguration of large numbers of machines has been discussed in [Manning93, Riddle94, Shaddock95]. "Forklift" upgrades of new hardware [Harrison92] share some but not all of the problems we faced in our move. Implementation of new networking topology without the problems or schedules imposed by physical relocation has been discussed in [Limoncelli97].

We believe our work is unique in requiring all these tasks to happen on a large scale in a relatively short time. We were allocated only one workday in addition to a weekend to shutdown and relocate our computing environment. We were expected to have a fully functioning network at our new location the following Monday. Ordinarily the complete reconfiguration of a network this size would be a challenge in itself. For our project, we had to account for the time required to disconnect and pack machines, load them into trucks, transport them across town, unload, and reconnect them at the new building. As we will detail, the resulting window of time available to handle the reconfiguration of all these machines was very small.

Introduction

The California Microprocessor Division of AMD (hereafter referred to as CMD) uses a large computer network of nearly 1000 nodes. Of these, over 600 are Sun and HP UNIX workstations and servers, primarily high-end dual-cpu UltraSparc machines. Most of the remaining nodes are NCD X-terminals and x86 compatible machines running Linux and Microsoft Windows. The majority of the CPU power resides in our server room on our headless compute server ranch. User desktops are mostly older Sun Sparc20 and HP 715 workstations or NCD X-terminals. Over three terabytes of disk space are served by six Auspex file-servers and one Sun E4000 with two RSM2000 chassis attached.

Most of CMD's engineers were originally employed by NexGen, which had been purchased by AMD in 1996. Our division was located in the former NexGen building in Milpitas. There had always been a desire by upper management to move CMD to the Sunnyvale main AMD campus to better facilitate interaction among the various groups within the company. As CMD began to reach the limits of its building capacity, the move from Milpitas to Sunnyvale became reality.

The engineers from the Milpitas building were to be relocated to a recently vacated AMD building (hereafter referred to as building 920) in Sunnyvale. We determined there was insufficient space to house CMD's rapidly growing server room in this building and therefore opted to locate CMD's compute and file servers in a new data center to be constructed within an existing facility (building 915) across the street from building 920. Having the server room in a different building required remote monitoring equipment as described later in this paper.

Our system administration team consisted of seven members with primarily UNIX expertise and two members who focus on PCs. Ideally a team of system administrators could be dedicated to move related tasks, but unfortunately the rest of the division was unwilling to forgo support while we prepared for the move. While preparations for the move were a high priority, it was definitely only a part-time job until the move date actually arrived.

The project required an large influx of contract personnel to handle the more routine tasks required of the move, thereby freeing the regular system administration team up to perform the more involved tasks and to handle troubleshooting. The physical move of

goods from Milpitas to Sunnyvale was covered by our facilities department in the scope of the entire office move. We arranged with an outside contracting company to provide additional manpower on the weekend of the move. Their scope of work included project management of the network installation, purchase and installation of racks for the network and servers, the disconnection and reconnection of machines, a pre-move network audit, and additional people to help troubleshoot individual machine and network problems.

Our department hired our own contractor to provide project management and additional muscle. One team was responsible solely for disconnecting the machines in Milpitas and reconnecting them in Sunnyvale. Several contract systems administrators were brought in to help debug individual machine problems.

Planning

Planning for the move began approximately four months before the move date. Our first task was to define the scope of the project. In typical use, all of our UNIX CPUs see continual use on a 24 hour by 365 day schedule [Cha98]. We had not had a complete planned systems and network shutdown in over four years of operation. It was therefore tempting to incorporate a plethora of major system changes from our todo list that had accumulated over the years.

However, the amount of downtime we had available was severely limited by our tight schedule. We also had to balance the scope of any major changes against the risk of changing too many things at once. From past experience we knew that it was essential to allow at least one extra day to account for any unexpected problems and to tie up any loose ends.

We had to determine which tasks and system changes we would tackle, and which we would postpone until our next downtime (i.e., never). The merits of each potential change were weighed against the implementation time required and the overall risk to the project. We settled on the list presented in Figure 1. Inclusion or exclusion was based on several factors:

1. How much risk does it add to the project?
2. How much value does it add to the resulting environment? What are the consequences of not doing the proposed change?
3. How difficult would it be to do if postponed? Does it require a complete shutdown?
4. Is there sufficient time to complete the task?

There were many interdependencies among the included tasks. The new network design required new network interface cards (NICs) on the Auspex and the client machines. The new NICs required updated operating system installations on the Auspex servers and many of the client machines. The topology of the new network (detailed later in this paper) required Fast EtherChannel on the Auspex file servers. All these tasks were deemed essential to having a scalable network in place to accommodate future growth.

Changing the NIS domain name added relatively little value to the project, but added almost no risk and would be somewhat cumbersome to do at any other time. The excluded tasks required too many changes on both the part of the system administration team and the user community. The two tasks that were partially completed were deemed important, but failed to satisfy the fourth requirement in that we simply ran out of time.

Included tasks:

- Construction of new data center in building 915
- Implementing a new network infrastructure
- Changing all client CDDI NICs to 100baseT
- Upgrading the oldest Auspex (antigen) to a more modern chassis.
- Upgrading Auspex servers to 1.9.2 OS
- Changing all NIC cards in Auspex to full-duplex 100baseT
- Implementation of Fast EtherChannel in Auspex servers
- Implementation of terminal servers for remote console access
- Changing NIS domainname from systems to cpgea.amd

Excluded tasks:

- Eliminating filesystems mounted by direct automount maps
- Rearrangement of legacy filesystem organization
- Replacing Sun's automounter with the publicly available amd automounter

Partial tasks:

- Changing all client 10baseT machines to 100baseT
- Upgrading all client OS to a standard level

Figure 1: Move tasks.

Pre-move Tasks

In order to make the move and reconfiguration happen within the limited time available, it was critical that we perform as many tasks as possible in advance of the move date. We were able to take small groups of machines out of production in order to install new network interface cards and perform clean operating system installations. Having a consistent stable installation ensured that all machines would boot properly without encountering unusual problems that could have resulted from undocumented or erroneous changes to the machine through the years.

Unfortunately, critical machines such as the file-servers, license servers, and many of the more powerful compute servers could not be taken down prior to the move date. Upgrades to these machines had to be accounted for in the master move schedule.

All machines were labeled one week prior to moving. To avoid problems of illegible writing and inconsistent nomenclature, we used laser printed labels and distributed them to everyone involved. A sample label is shown in Figure 2. These printed labels were attached to larger color-coded moving company labels. Each color corresponded to a different zone in the new building to facilitate location of the equipment in the new building. The printed color-name on the label helped ensure that the right color label would be used.

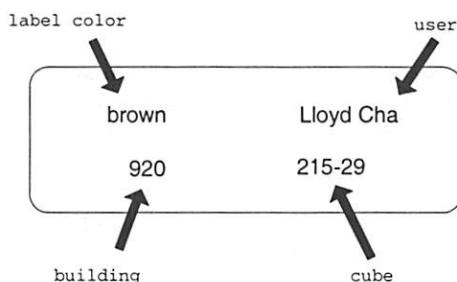


Figure 2: Sample label.

Typical moving labels are designed so that they can be peeled off furniture and the like after the move without leaving much residue. Unfortunately, we found that they stuck poorly to the cases of most of computer equipment. We ended up running around with rolls of clear tape to make sure that nothing fell off during the move.

The task of labeling the computers also provided a check to see what machines may have been missed in our master move list. In addition, we had included a detailed hardware audit in the scope of work that our contractor was to perform.

Unfortunately, the resulting audit turned out to be woefully incomplete. Due to some miscommunication, the audit was performed by starting with a list of

machines generated from our DNS database and then ascertaining their locations. While this did help qualify the DNS database, it did not perform the essential function of identifying hardware that had somehow been missed in the DNS listing. These machines included those that had returned from repair and had swapped names, machines that had recently been acquired, and machines operating without proper DNS registration (mostly renegade PCs).

We should have insisted on an audit that would physically cover every square foot of our building in a systematic way to identify every piece of computer hardware attached to our network. This would have allowed us to double-check our existing machine databases, identify archaic hardware that we should dispose of, and bring renegade machines to justice. This would also have given us a definitive checklist to go by to ensure that nothing had been missed by the movers.

Networking Considerations

Our existing network was not optimally configured as a result of extremely rapid growth with little available downtime. On several occasions over 100 machines at a time were added in batches with minimal disruption to the existing production network. This resulted in an irregular and poorly balanced network. Budget and downtime limitations prevented us from overhauling the entire network, which resulted in a hodgepodge of modern and outmoded equipment. Our backbone was a hybrid of ATM and FDDI, and client machine network interface cards (NICs) included 10baseT, 100baseT, CDDI, and FDDI.

Our options during the move were either to build a new network from newly purchased equipment and stage it several weeks before the move, or to tear down our old network the weekend of the move and use the components as a basis for building the new network. We were able to convince management that pursuing the latter option was too risky and would require too much downtime. We were able to justify the cost impact by noting that our older network equipment could be re-deployed to upgrade other even older equipment in use by other divisions within our company.

Changing technologies and the opportunity to build a scalable network from scratch resulted in a vastly different network design from our existing network. CMD's powerful workstations and implementation of LSF [Platform97] for load sharing demanded high-throughput between all machines with a particular emphasis on NFS traffic. Our new network is based primarily on 100baseT, with extensive use of Cisco's new FastEtherChannel technology [Cisco97, Cisco98a] for interswitch and fileserver [Auspex98] links. Further information about our new network topology can be found in [Cha98].

Our New Data Center

This move gave us the opportunity to construct a new data center in building 915 with many features that would improve our efficiency in maintaining the network. A large UPS system that covered the entire server room replaced a hodgepodge collection of single-system UPS units that were scattered around our existing data center.

Our old server room configuration had keyboards attached to every machine and a terminal that was wheeled around on a cart to attach to the console port of any machine that required attention. This was both messy and inefficient. For our new data center, we opted to use Xyplex 40-port terminal servers to provide remote console access. We reserved one port to possibly connect a modem for access to the machine consoles in the event that the network was completely down. Details of our implementation are available in [Cha98].

Our Sun servers automatically use the serial port if a keyboard is not detected upon bootup. One problem with Sun servers is that a break signal is perceived by the host if the serial port is connected to a terminal server that has been power cycled. Unfortunately, the team testing to see that the terminal servers would boot properly when power cycled was not aware of this fact and managed to simultaneously halt every machine connected to the terminal servers during their testing.

Fortunately our terminal servers are protected by the same UPS system that protects the Sun machines, so this problem should not be an issue except in case of operator error. As of this writing we have not put much effort into implementing a solution, however some suggestions are available in [Cisco98b].

The "100 Server" Move

We were permitted downtime on one hundred of our compute servers one weekend in advance of the primary move date. These machines were already located in Sunnyvale in a room adjacent to our new data center due to a lack of available space in Milpitas and were therefore not covered in the scope of our physical relocation.

We were taught an early lesson as to why extensive testing of scripts is essential before widespread implementation. The reconfiguration script had been only proofread but had not been extensively tested due to lack of time. It would have been time well spent. Our reconfiguration script suffered from two embarrassing and critical typos. The first problem was that the new default route was incorrectly entered, making the machines unreachable from anything but the local subnet. The second problem was that the NIS domain-name was misspelled, which caused every machine to hang at waiting for an NIS server to become available.

What's worse, these errors caused name service to fail on these clients rendering the .rhosts file useless. To avoid having to write an Expect script to login to each of the machines, we temporarily stuffed our root passwords into a .netrc file so that we could use rexec to run our "cleanup" script to fix all the errors. We also added at least one IP address to each of our client .rhosts files so that we would not be stuck with this again.

This experience taught us that it's worthwhile to run all procedures on one machine in "live" mode no matter how far behind schedule we were. No amount of proofreading is an adequate substitute for actually running the script or procedure and testing the results.

Schedule Overview

Our overall schedule looked like this:

Thursday 6 pm:

All users logged off. Any jobs still running may be terminated. Shutdown script run on all desktop machines. Desktop machines powered off and packed.

Thursday 9 pm:

Shutdown scripts run on all compute servers.

Friday 3 am:

Final full backup before move for all file servers complete. File servers shutdown and packed for moving.

Friday 6 am:

File servers begin loading onto moving vans, followed by compute servers and desktops.

Friday 9 am:

File servers arrive in Sunnyvale and are unloaded and placed in new location for reinstallation. Auspex begins reinstallation and required hardware changes.

Friday noon: Hardware changes to file servers complete. Compute servers and desktops begin to arrive at new building.

Friday 6 pm: Operating system upgrades to file servers complete. File servers should now be fully operational. Compute servers and desktops should all be in place. Reconnection of compute servers and desktop machines begins.

Saturday 6 am:

All compute servers and desktops operational. Exceptions to be noted and addressed by systems admin. Global script run to check health of all machines.

We opted to set a very aggressive schedule and allow sufficient time for slips in the schedule rather than setting a relaxed schedule with very little room for error. Had we following a relaxed schedule, we would run the risk that one of the tasks in the late stages would slip leaving little room for recovery.

That was our plan. The actual timeline of events which we will detail below turned out to be somewhat different.

Shutting Down

Our shutdown scripts had two main objectives: reconfiguration and shutdown. First, the shutdown script modified each machine's startup files such that on power up the system would be configured for the new network. At a minimum the IP address, netmask, and NIS domainname would need to be changed. Some machines would require new network interface cards. After applying all these changes, the scripts would 'sync' and 'halt' each machine in preparation for being powered down and disconnected.

A single master reconfiguration file drove all the IP address changes. All the shutdown scripts consulted this file, as did the script that we used to generate our new DNS source files. Using a single master file allowed for easy insertion of last minute changes and also ensured that the local host files and DNS database would be consistent.

Our original plan called for deploying the reconfiguration scripts to the local /tmp disk of every machine. We would then start a process that would run in an infinite loop, waking up periodically to look for the local hostname in a central control file. Once the

```
#!/bin/csh
# Script for halting a machine
#
if (-x /usr/local/bin/get_arch) then
    set archtype = '/usr/local/bin/get_arch'
else
    set archtype = "unknown"
endif
switch ($archtype)
    case 'sunos':
        /usr/etc/halt >& /dev/null
        exit 0
        breaksw
    case 'solaris':
        /usr/sbin/halt >& /dev/null
        exit 0
        breaksw
    case 'hp9':
        /usr/sbin/shutdown -h -y 0 >& /dev/null
        exit 0
        breaksw
    case 'hp10':
        /usr/sbin/shutdown -h -y 0 >& /dev/null
        exit 0
        breaksw
    case 'linux':
        /sbin/halt >& /dev/null
        exit 0
        breaksw
    default:
        exit
endsw
```

Figure 3a: Sample shutdown script.

```
#!/bin/csh
# Wrapper script for shutting down machines

/usr/local/admin/move/scripts/halt_sys >& /dev/null &
exit 0
```

Figure 3b: Sample wrapper for shutdown script.

hostname appeared, the reconfiguration script would be launched out of /tmp and the machine would shut itself down.

Due to last minute changes in the reconfiguration scripts, we did not have enough time to distribute the changed scripts to each machine. The fallback was to rsh to each machine using a simple foreach loop and call the reconfiguration script from a central location. The problem with this approach is that some machines would hang on the rsh during the shutdown call, and the foreach would not continue. A more robust solution is to rsh to the machine with a call to background the actual halt script using csh, which would allow the foreach to continue. Sample scripts are provided in Figure 3.

Once the shutdown scripts executed, teams of contractors moved through the building powering down and disconnecting machines. Peripheral components such as keyboards and mice were collected into labeled clear plastic bags and were left with the machines for pickup by the movers.

Bringing Up the Fileservers

The critical path in our move schedule centered around the move and reconfiguration of the Auspex servers. They were the last machines to be shutdown in the old building and the first that would be required in the new location. In addition, substantial hardware changes were required to support the new network hardware.

We hired Auspex to handle the packing and unpacking of our Auspex servers. Packing materials designed specifically for the equipment we were moving were shipped well in advance to our old address and staged in preparation for the move. We completed a full level 0 backup just before shutting the file servers down. Everything had gone smoothly up to this point.

The disassembly of the Auspex servers took longer than expected, and the movers were several hours late in delivering the goods to the new location. It appeared that both our moving company and Auspex had underestimated the amount of manpower needed to complete the tasks required in the allotted time. This put us several hours behind on the most critical path of the schedule. We encountered additional delays in getting the new hardware installed in the Auspex servers required to support our new network. By midnight Friday night, the file servers still had not been powered up. By this time most of the vendor technicians and support engineers who had been working since 3am the previous night were exhausted. Powering up the servers brought more unpleasant surprises. Three of the seven servers failed to boot due to improperly installed hardware. We spent most of the night debugging server problems.

The one Auspex server that we were upgrading during this move was an exception to the usual

process. This was a "forklift-style" upgrade in which the new server was delivered with the old server being returned for credit at a later date. The new server was ready by the move weekend awaiting the final step of copying over 300GB of data from the old server to the new. We considered several methods:

1. Pre-load the new server from the last full backup (one week before the move). Run an additional incremental backup on the old server before shutting down and apply it to the new server on the weekend of the move.
2. Pre-load the new server by copying data over the Milpitas-Sunnyvale WAN (OC-3). During the move, use rdist to update any files that have changed.
3. Do nothing before the move. During the move, attach both the old and new disk chassis to the new server and copy the data locally from disk to disk. This can be done rapidly on the Auspex architecture by using the dedicated storage processors and bypassing the UNIX kernel altogether.

We had initially attempted to use option 1, but our efforts were derailed due to issues with the backup software and licensing that severely impacted performance. Option 2 proved to be impractical due to long transfer times over the WAN link. We ended up using option 3, the direct transfer from disk to disk.

Option 3 provided the fastest and most foolproof method of transferring the data but had a critical disadvantage that it would have to be completely done during the move weekend. Unfortunately the process also required a lot of manual work by the operator, some of which could probably have been scripted in advance although we failed to do so. The hardware and disk swaps necessary to the procedure could not be automated anyways. The entire procedure became the single-most time consuming task of the entire weekend. The complete copy did not finish until sometime Saturday afternoon.

Lessons learned:

- Be sure that adequate manpower is available, with additional help available to be called in if needed. Overworking the move team is unwise as it increases the probability of mistakes.
- Script out and document changes in detail. Even the simplest procedures become error prone when operating under pressure or sleep deprivation. Leave nothing to chance.
- Be realistic about schedules and have a contingency plan in place should things fall far behind.
- Don't drive yourself home after staying up for 30 consecutive hours.

Bringing Up the Client Machines

Our general plan was for contractors to oversee the routine powering up and booting of client

machines. They would login to a test account which would run a script that would test for basic functionality. Exceptions would be flagged and handled by our system administration team.

The number of system administration problems encountered in bringing up client machines was overshadowed by logistical problems related to the physical move itself. Our moving company had underestimated the amount of manpower necessary to relocate the goods while keeping to our schedule. Many machines were delayed in their arrival, and those that did arrive were often placed incorrectly despite being properly and clearly labeled. System configuration problems were rapidly fixed, but valuable time was lost scouring the building for misplaced monitors, keyboards, and other parts.

Results

Despite our schedule setbacks, we were able to provide a fully functional computing environment for our users by Monday morning. Our LSF reports (our LSF setup is discussed in [Cha98]) indicated that fewer than 5% of the machines were unavailable on Monday. None of these machines was critical and the majority of these were due to circumstances somewhat beyond our control (e.g., hardware failures, lost peripherals, etc). While we were happy with the end results, we realize that many of the lessons learned from our experience could have saved a lot of extra effort and stress. We hope that readers of our paper will be able to benefit from both our successes and failures.

Suggestions for a Successful Move

In this section we provide additional tips and techniques that we have learned from our experience that were not already covered above. Some of the suggestions reflect things that worked well for us, while others were mistakes we learned to avoid. Several are common sense suggestions that could easily be overlooked in the rush. In no particular order:

- Keep copies of essential data related to the move and reconfiguration in a central location. Don't forget to copy them to an external or stand-alone system before powering down since you won't have access to them while your network is in transit.
- Make a priority list and make sure everyone on the project has a copy. Solve problems that affect the most systems first. Document dependencies (eg. NIS server needs to be available before NIS clients are booted). Much of this seems obvious, but remember that in a reasonably stable environment these requirements are often taken for granted.

Typically file servers, DNS servers, and NIS servers should be attended to first followed perhaps by the mail server, license servers, and

web servers. In our particular case, the LSF master server [Cha98] was also a high priority since we used it to collect data on the general health of our computing cluster.

The priority list should also take into account the relative importance of the various services provided to the users. If it becomes impossible to have everything available on schedule, you will at least want to have the essential functions available.

- Have a healthy stock of basic office supplies and deploy them in accessible places at your new location. You want to avoid wasting time looking for a pen and paper, and you certainly don't want important notes left untaken for lack of a writing instrument. Permanent markers are invaluable.
- Communication is essential. Pagers, cordless phones, and walkie-talkies are all useful although be aware that some older computer hardware may be sensitive to radio transmissions. Setup a command center with a whiteboard or bulletin board in a visible area that is kept up-to-date with the latest status and whereabouts of key personnel.
- Arrange a blanket purchase order or other means of purchasing necessary items that may have been overlooked in earlier planning. We were fortunate to be located near a large Silicon Valley electronics store that was able to handle our emergency supply needs.
- Provide meals and snacks on-site. This keeps critical people from wandering off to lunch or dinner and becoming unavailable for consultation.
- Laptops are invaluable for storing information (IP lists, phone lists, etc), connecting to outside systems, substituting for dumb terminals, downloading patches and licenses, testing the network, etc.
- Script and automate as much as possible. Document routine procedures that cannot be scripted. Spread knowledge around as much as possible.
- Label label label! Audit and inventory all items. Label individual components and SCSI ID and DIP switch settings that may get inadvertently bumped during transportation. If possible, have the audit performed by someone familiar with the hardware involved to prevent headaches caused by improper or ambiguous information.
- Test whatever can be tested. Stage a dry run or preliminary move if possible.
- After the move, provide information to users on whom to contact for problems. Setup a hotline to coordinate trouble requests if one doesn't exist.
- Cleanup as much as possible before the move!
- Give your users incentive to co-operate. Get

support from management to send them to a movie or other company sponsored events to keep them out of the way. Set reasonable expectations.

Availability

Please contact the authors at <lisa98@cmd-mail.amd.com> regarding availability of scripts referenced in this paper.

Author Information

Lloyd Cha is a MTS CAD Design Engineer at Advanced Micro Devices in Sunnyvale, California. Prior to joining AMD, he was employed by Rockwell International in Newport Beach, California. He holds a BSEE from the California Institute of Technology and a MSEE from UCLA. He can be contacted by USPS mail at AMD, M/S 361, PO Box 3453, Sunnyvale, CA 94088 or by electronic mail at <lloyd.cha@amd.com> or <lloyd.cha@pobox.com>.

Chris Motta is the manager of the CMD Systems and Network Administration department. He holds a BSME from the University of California at Berkeley. He has held a variety of systems administration positions including UNIX and networking consulting. Electronic mail address is <Chris.Motta@amd.com>, and USPS mail address is M/S 366, PO Box 3453, Sunnyvale, CA 94088.

Syed Babar received his master's degree in computer engineering from Wayne State University in Detroit, Michigan. He works at Advanced Micro Devices in Sunnyvale, California as a Senior CAD Systems Engineer. He can be contacted via e-mail at <Syed.Babar@amd.com> or <Syed_Babar@hotmail.com>.

Mukul Agarwal received his MSCS from Santa Clara University. He joined Nexgen, Inc. in Milpitas, California as a CAD Engineer in 1993. He switched to systems and network administration in 1995 and has been a System/Network Administrator ever since. Reach him via e-mail at <mukul.agarwal@amd.com>.

Jack Ma holds a BSCS from Tsinghua University and a MSCS from Computer Systems Engineering Institute. He was a UNIX software developer at Sun Microsystems before joining Taos Mountain at 1995, where he now works as a networking/UNIX system consultant. He can be reached electronically at <ylma@netcom.com>.

Waseem Shaikh holds a master's degree in computer engineering from the University of Southern California and received his bachelor's degree in electrical engineering from University of Engineering and Technology in Lahore, Pakistan. He was a System/Network Engineer at Steven Spielberg's Holocaust Shoah Foundation, a System Consultant at Stanford Research Institute, and is now working as a

System/Network Consultant with Taos Mountain. He can be reached at <shaikh@netcom.com>.

Istvan Marko is a self-educated Computer Specialist currently working as a System Administrator employed through Volt Services Group. He can be contacted via e-mail at <imarko@pacificnet.net>.

References

- [Auspex98] "Auspex Support For Cisco Fast EtherChannel," *Auspex Technical Report #21*, Document 300-TC049, March 1998.
- [Cha98] Cha, Lloyd, et al., "The Evolution of the CMD Computing Environment: A Case Study in Rapid Growth," *LISA XII*, Boston, MA 1998.
- [Cisco97] Cisco Systems, Inc. *Fast EtherChannel*, Cisco Systems Whitepaper, 1997.
- [Cisco98a] Cisco Systems, Inc. *Understanding and Designing Networks Using Fast EtherChannel*, Cisco Systems Application Note, 1998.
- [Cisco98b] Cisco Systems, Inc. *Terminal Server Break Character on Cisco Access Servers*, Cisco Systems Field Notice, April 21, 1998. URL: <http://www.cisco.com/warp/public/770/fntsbreak.html>.
- [Harrison92] Harrison, Helen E, "So Many Workstations, So Little Time," *LISA VI Proceedings*, Long Beach, October 1992.
- [Limoncelli97] Tom Limoncelli, Tom Reingold, Ravi Narayan, and Ralph Loura, "Creating a Network for Lucent Bell Labs Research South," *LISA XI Proceedings*, San Diego, CA, October 1997.
- [Manning93] Craig Manning and Tim Irvin, "Upgrading 150 Workstations in a Single Sitting," *LISA VII Supplementary Materials*, Monterey, CA, November 1993.
- [Platform97] Platform Computing, "AMD's K6 Microprocessor Design Experience with LSF," *LSF News E-mail Newsletter*, Platform Computing, August 1997.
- [Riddle94] Paul Riddle, "Automated Upgrades in a Lab Environment," *LISA VIII Proceedings*, San Diego, CA, September 1994.
- [Shaddock95] Michael Shaddock, Michael Mitchell, and Helen Harrison, "How to Upgrade 1500 Workstations on Saturday and Still Have Time to Mow the Yard on Sunday," *LISA IX Proceedings*, Monterey, CA, September 1995.

Anatomy of an Athena Workstation

Thomas Bushnell, BSG and Karl Ramm – MIT Information Systems

ABSTRACT

This paper presents work by many developers of the Athena Computing Environment, done over many years. We aim to show how the various components of the Athena system interact with reference to an individual workstation and its particular needs. We describe Hesiod, Kerberos, the locker system, electronic mail, and the software release process for workstation software, and show how they all interrelate to provide high-reliability computing in a very large network with fairly low staffing demands on programmers and systems administrators.

Overview

The Athena system is an integrated campus network of Unix workstations for academic computing at MIT, comprising thousands of workstations and hundreds of servers. We manage Athena with a fairly reasonably sized central staff. Workstations are located in public clusters managed by the central staff, in private clusters maintained by various academic departments, in faculty or staff offices, in public hallways and lobbies, in dormitories, in libraries, and laboratories. Most of these computers are available to any member of the MIT community (it is in this sense that we use the term “public” in this paper – MIT does not provide any computing facilities for the general public).

An Athena workstation is a typical Unix workstation which can provide a platform for users to run their applications: sending and receiving email, running courseware, text editing and formatting tasks, and so forth. Much that an Athena workstation does is done locally as on any other Unix computer. Much is provided by contacting servers over the network. Servers do not trust the integrity of the workstation or its software which thus enables users to run their own Athena workstations, and us to publish to the entire Athena community the root password for public workstations.

In this environment, Athena handles most security issues by the strategy of serial reuse: a given user logs in, and has full and total control over the workstation. Then she logs out, the workstation cleans itself up in preparation for another user, and then waits until another user logs in and has total control. We do not attempt to fully address simultaneous-use problems, except in special cases.

Because Athena workstations are located in such a diverse array of places, and because there are so many of them, they are managed in a way to require almost no intervention from the cluster maintenance staff. A workstation requires a certain ineradicable amount of hardware support. Beyond that, it requires

typing a few lines into a PROM monitor and perhaps inserting a floppy to install a workstation, and from then on it will not only install itself, but also automatically update itself for years as the Athena system develops and changes, all without needing any manual intervention.

The purpose of this paper is to describe some of the key Athena services that hold this arrangement together, from the standpoint of an Athena workstation and its maintenance. Many important Athena features, such as the Zephyr messaging service or the extensive courseware developed at MIT, are not described here.

Hesiod

Key to the operation of the Athena environment is the Hesiod directory service [Hesiod]. Hesiod provides directories for many different things, such as filesystems, users, printers, and post office servers. Hesiod lookup is done through the Domain Name System, by requesting TXT records. Originally we used the separate class HS, but now we store the records in the IN class. In this way, large databases need not be replicated across many machines, which significantly reduces administrative overhead and risk.

For example, each user in the Athena environment has a password file entry stored in Hesiod, as shown in Example 1. (Note that the actual password is not stored in Hesiod; user authentication is provided by the Kerberos system, as described below.)

Each of the tables managed by Hesiod has a name; the one above is the ‘passwd’ table. Each user also has a post office box assigned on one of many possible POP servers; the ‘pobox’ table in Hesiod is used to determine on which post office server a given user’s mail should be found. For example, the ‘pobox’ entry for one of us is:

```
POP P08.MIT.EDU tb
```

Programs which need to read the database use library functions which issue properly formatted DNS requests to the normal DNS servers, which respond

```
tb:*:7722:101:Thomas Bushnell BSG,,E40-342d,31368,6230654:/mit/tb:/bin/athena/tcsh
```

Example 1: Sample Hesiod password file entry.

with the requested entry. The libraries then dissect the DNS reply and return the desired entry.

One disadvantage of this mechanism is that many user programs must be modified to issue Hesiod library calls at the proper places. For example, any program that fetches mail from the mail spool must have code added to request the identity of the post office server from Hesiod. In some cases this problem can be ameliorated: the Athena login process, for example, fetches the password file entry for the user at login and temporarily stores it in the password file on the local workstation, so that other programs can simply call `getpwnam` and have it work.

Very similar functionality is provided by the NIS service designed by Sun Microsystems. However, Hesiod is different in several important respects. First, and perhaps most importantly, it uses the very well-tested DNS infrastructure, including caching, and does not depend on the broadcast characteristics of networks as NIS does. NIS is generally used to handle information which needs to be secure, but in the Athena environment, Hesiod is never used for such information and other services are responsible for security.

NIS databases can be completely downloaded; this ability is fundamental to the operation of NIS. However, Hesiod never depends on this functionality, and BIND makes possible various kinds of discrimination on what kinds of zone transfers should be permitted. (Because of privacy considerations, we do not permit arbitrary downloading of the Hesiod tables.) Hesiod can be easily extended to support new database types with little effort, but NIS maps are basically limited to the default set.

The current implementation of Hesiod limits responses to 1K bytes, but nothing in the actual protocol has such a limitation. One serious disadvantage, however, is that Hesiod cannot be updated dynamically. NIS also has problems in this area. As a consequence, MIT generates the Hesiod databases once a day, and updates them overnight.

Hesiod will automatically be able to benefit from DNSsec and once a standard for dynamic DNS updates is approved (one is on the IETF standardization track now), Hesiod will be able to easily take advantage of it.

Despite some of these disadvantages, we have found Hesiod to function better than NIS could in such a large environment. Hesiod is never used for information that needs security. The reliability and scalability of DNS cannot be beat, and we have had no significant problems with limitations on record lengths or the delay in updating DNS databases.

Kerberos

Kerberos [Kerberos] is the system in Athena which manages authentication. It provides a global

space of identification names and a secure facility for mutual authentication using those names. Athena servers and workstations then use the Kerberos system to make authorization decisions.

A Kerberos name is composed of two parts, a name and a realm. The name is composed of multiple separate strings, but no interpretation is imposed by Kerberos itself on the contents and relation of those strings to each other. (In the older version 4 of Kerberos there were exactly two strings in the name, one called the "principal" and one called the "instance.") A realm identifies indirectly the servers and authorization scope of the name. Each system using Kerberos must maintain a table mapping realm names to the associated Kerberos servers.

Kerberos servers function by creating entities known as "tickets." A ticket is an encrypted data block specifying a Kerberos name, a time stamp, an expiration stamp, the Kerberos name of a service, and a secret key. By presenting the ticket to the named service, the service can be assured that the presenter has the claimed identity. Such services might be POP servers, file servers, telnet servers, and so forth. The ticket is encrypted using a key known only to the server, and the key contained in the ticket is also known by the client. In order to use a ticket successfully a client must also know the secret key, and so sniffing the ticket in transmission does not help an attack. A secret key together with its associated ticket is known as a "credential."

One important service is the Kerberos server itself. Tickets for the Kerberos server are called "ticket granting tickets." Possession of tickets for a file server allows you to use the fileserver, and possession of tickets for a post office server lets you retrieve mail (in both cases, of course, subject to further authorization on the server). Possession of a ticket granting ticket, however, lets you obtain tickets for any other service you desire. Users obtain ticket granting tickets when they log in, upon proving to the Kerberos server that they possess a correct password. From that point, the ticket granting ticket is used as necessary to obtain service tickets for the various servers the user needs.

These credentials (tickets and associated client keys) are conventionally stored in a file in `/tmp`. They are not stored in user home directories because it is important for the integrity of the system that they not be transmitted by a sniffable remote file protocol. The ticket file is kept protected using the normal Unix facilities; thus someone who has broken into the workstation might steal the tickets. (We generally avoid such problems by the serial reuse model described above.) Because tickets have a built-in expiration time, the window of risk is fairly well-known. When users log out, a special program called `kdestroy` is used to delete the ticket file, by writing null bytes to the file containing the tickets before unlinking it, thus guaranteeing its true destruction.

Because all Athena services are authorized based upon possession of proper Kerberos tickets, no service depends on the integrity of the user's workstation. Accordingly, the root password for public workstations is advertised to the entire Athena user community. Because many of the facilities of Unix are available only to the superuser, this greatly improves the utility of the Athena system, as well as improving its security. It should be noted that we are open to the thread of a malicious user modifying workstation software to capture passwords; we believe that this risk is inherent with having unsupervised hardware and we would not actually improve security by trying to hide the root password.

Shared workstations do exist, however, the most important being the public dialup machines. These have a secret root password, and are very carefully maintained and managed. Because the Athena model is based on serial reuse, these simultaneously shared machines must be given special attention. Some facilities are not provided on them, and Kerberos tickets on them are theoretically more vulnerable.

Lockers

In a substantial distributed computing environment, it can be problematic to have all the filesystems on the campus mounted on every workstation. Many remote filesystem techniques behave poorly when servers are unavailable, and many operating systems cannot handle a large number of filesystem mounts at a single time.

Accordingly, file space in the Athena environment is separated into "lockers" [ATPD], each of which appears as a distinct filesystem to users, but is not mounted all the time on each workstation. Each locker has a name, and the 'attach' command will look up the name and mount the filesystem in the /mit directory. (In special cases lockers may be mounted elsewhere.)

Each user home directory is a locker. A locker might also hold one or more third-party software packages, or anything else. Student organizations (even informal or transient ones) can generally get lockers created with little trouble, and manage and maintain them.

All file storage in the Athena environment is handled through the locker mechanism. Users mount lockers with the 'attach' command, and these lockers are generally unmounted when the user logs out.

Lockers which contain programs intending to be run are generally mounted by the 'add' command, which runs attach, and then adds the appropriate subdirs of the locker to the user's command execution

path. This frees users from needing to understand the ways that path search happens or the manner in which lockers organize the various binaries that have been compiled for the various supported architectures.

Lockers do not all use the same remote filesystem mechanism, making them more flexible than using NFS or AFS alone. Each locker can use whichever filesystem is most convenient, and users in general need not consider the difference.

Locker names are stored in a Hesiod table named 'filsys'. Some sample Hesiod records for lockers are shown in Example 2. These specify the filesystem protocol type, where the locker is to be found, how to mount it (writable in these examples) and where it will be mounted on the local workstation.

Each locker of course is associated with some things not covered in the scope of this paper, such as disk allocation issues on servers, backup policies, and the like. All these are hidden behind the locker abstraction, and generally need not be noticed by users at all.

Email

Electronic mail on Athena follows the now fairly standard pattern of central mail servers accessed via POP (or some other mail-specific protocol such as IMAP). Rather than a single large POP server, Athena has several POP servers and a Hesiod table (called 'pobox') is used to identify the correct POP server for a given user.

Athena workstations do not normally run sendmail daemons, but do run sendmail on demand to handle outgoing user mail. All outgoing mail is forwarded to a local mailhub, where it is immediately delivered or queued. In this way we avoid maintaining queued mail on the clients as much as possible. Sometimes mailhubs are too loaded to accept mail or are even down, in which case mail does end up being queued on the local workstations, so workstations have a periodic cron job to deliver any queued mail.

Reactivation

Each workstation periodically runs a script called 'reactivate' when no one is logged into it, which runs various housekeeping functions. The goal of the reactivation procedure is to improve the hands-off maintainability of workstations, relying on them to keep themselves in a sensible state as much as possible.

Much of what reactivate does is ordinary cleanup duties. Any stale server state, for NFS, or AFS, or the Zephyr notification system is deleted or synchronized as necessary. Stray processes are killed and attached

```
AFS /afs/athena.mit.edu/user/t/b/tb w /mit/tb
NFS /ul/bitbucket JASON.MIT.EDU w /mit/bitbucket
```

Example 2: Sample Hesiod locker entries.

lockers are detached. Special lockers that are always to be in place, called the "system packs" are reattached to make sure the correct ones are mounted in the correct locations. On public workstations, the password file is reset to the default and many local files are checked to make sure they match the prototype files for public workstations.

One very important function provided by the reactivate script is to check whether a software update is necessary for this workstation and to schedule it.

System Packs

Originally, Athena maintained its own operating system, a variant form of BSD 4.3. This was stored in a special locker known as the "system pack" and mounted on /srvd. Over time, this broke down, for two central reasons: first, maintaining an operating system is a complex and time consuming task, and as new hardware came and went it became attractive to use the vendor operating system rather than porting our own. Second, various third party software became attractive to various members of the MIT community. Much of this software is distributed only in binary form for the various vendor operating systems; using an Athena-specific operating system would complicate greatly the use of such software.

We therefore now use the standard vendor operating systems, and what used to be the srvd pack is now split into two parts. One is the os pack, mounted on /os, and the other is the srvd pack, on /srvd. The os pack contains an essentially unmodified copy of the vendor operating system. (We do need to make some modifications. Currently the packs are stored on AFS, which can't do hard links properly, so those must generally be turned into symlinks. Also, some security and permission fixes must be made in the os pack.)

The srvd pack contains all the Athena-specific software that we build ourselves. Some of it contains programs often found in other vendor operating systems, but for which we need to have a single standard version, or because we have added special local modifications. Also the srvd contains Athena-specific programs, such as Hesiod- and Kerberos-capable versions of mail-reading software.

Workstations' local disks contain some software locally, but much is accessed through symlinks into the /srvd or /os packs. The decision about which goes where is made separately from the organization of the packs themselves, which contain the complete system, including those parts normally found on the workstations' disks themselves.

Workstations determine which system packs to attach via Hesiod. At any given time, for any given architecture, we will usually have packs for two minor versions of the current major version; one in testing and one that is deployed widely. We also keep old versions online for a reasonable approximation of forever (the oldest kept online right now is from 1989).

A third system pack has recently come into use on some platforms, the /install pack. This locker contains the raw OS install contents, for example, the inst packages for Irix or the tar files for NetBSD. This locker is not used except as a resource during the update or install procedure.

Release Cycle

Most software in the Athena computing environment is available through lockers. Each locker may be maintained and released differently, and no coordination is inherently necessary. However, much software is either located directly on each workstation, or is so central to the environment, that it must be released in a coordinated manner. For this software, we have a careful release cycle mechanism in place [RelEng].

Full-blown releases take place once a year, but smaller patch releases are made from time to time as necessary. As the name suggests, patch releases are generally intended to solve minor problems or security concerns that have developed. Major changes (for example, the upgrade from Kerberos IV to Kerberos V) are reserved for the major release.

The software that is part of the release comprises two main categories: the operating system, and Athena-specific modifications. Operating system updates take place only in the major releases, and require a great deal of care and attention. We currently support Solaris and Irix, and for each we have a designated engineer who has primary responsibility for that operating system's installation and functioning.

In addition, we have a release engineer who monitors the work of the operating-system-specific work and also manages the Athena-specific code. Many developers are able to modify this code as necessary, but having a single release engineer responsible for its overall function and for the coordination of the actual release process enables more consistent management of the release process.

A team meets weekly to discuss release issues and changes to the release; this team includes not only the release developers, but also members of the server operations team, and user support people. The major releases are carefully coordinated to academic schedules, server needs, and administrative requirements, and require extensive public documentation (posters, web pages, and so forth) describing the new features and changes associated with the release.

Update and Install

The process of updating a workstation begins with the workstation itself. Once a new release is available, the workstation's reactivate script notices via Hesiod that it should be running a new release, and it picks a time to actually execute the update. Updates do not occur immediately upon noticing that one is available in order to avoid swamping file servers and networks, or the specter of all available workstations

being busy updating, and a user not being able to log in. And of course, an update is taken only when the workstation is otherwise idle.

An update begins by mounting the system packs corresponding to the new release. If the new release involves no operating system upgrade, then the procedure will be fairly simple: new files as necessary are copied from the new `srvd`, and the workstation is rebooted. If an operating system upgrade is necessary, the procedure will be much more complex, because new shared libraries and kernels must be installed in a careful manner. In either case, however, the procedure is the same from the workstation's standpoint. Upon mounting the new system packs, a script in those packs is run which takes over and manages the entire update automatically.

A key goal of the Athena system is to provide hands-off maintenance and the orderly update procedure is a major part of that. Workstations update themselves without needing any manual intervention. In major updates, a certain number of workstations fail the update procedure for various reasons and do require manual intervention, but this is a tiny percentage of those which upgrade themselves without a hitch.

Installing a workstation is a similar procedure. A few commands are typed to the PROM monitor. The workstation is booted over the network and completely automated scripts handle everything else to install the workstation. In this way a large cluster of new workstations can be installed with very little typing necessary.

The creation of the update and install scripts often requires some subtlety. But nothing is seriously complex beyond understanding, and the entire process is the only one feasible for an environment this large. We have workstations scattered across campus, not only in general clusters, and the labor of having to attend to each one individually in an update (even if only to type a few commands) would be exceedingly tiresome.

Customized Workstations and Servers

The typical Athena workstation lives in a publicly accessible cluster and manages itself as described above. Such workstations should not be customized – to do so would defeat the uniformity expected of each workstation by Athena users. Accordingly, the periodic reactivate process attempts to notice customization, and if it detects it, then the customization is reversed by recopying the modified files.

However, for many Athena computers this would not be appropriate. Such are considered “private” workstations and may be customized at will by their owners. A private workstation might live on a user's desk, be part of a specialized departmental cluster, be a server of some sort, or in some other fashion need special treatment. Sometimes a private workstation

has no customizations beyond limiting who may log in to the computer.

Private workstations of course must have the customization-prevention measures disabled. Also, while a normal public Athena workstation can be centrally administered and needs no direct intervention to keep it running, a private workstation, if customized, will generally require more attention.

Automated customizations are made possible through a tool known as ‘`mkserve`’. Perhaps the most frequent customization on a private workstation is ‘`mkserve remote`’ which enables remote login access to the workstation via telnet, rlogin, or ssh.

Athena servers are also set up with the `mkserve` customization mechanism. (In fact, this is reason for the program's name.) An NFS server can be easily set up with ‘`mkserve nfs`’, a Hesiod server with ‘`mkserve hesiod`’ and so forth. With a hundred servers or so in the Athena environment having this ease of management becomes crucial.

The software which makes up these automated customizations is considered part of the Athena release. Generally the programs and data files the service needs are copied onto the local disk. The release update process also updates any customizations made through `mkserve`, which enables the great majority of customized workstations to benefit from the hands-off management style available for public cluster workstations.

Conclusion

Upgrading to a new version of the vendor operating system takes a fair amount of time. Because our release cycle is fairly rigid, significant delay might occur between the release of the operating system and our ability to get it into each workstation. Some of this time is imposed because of the complexity of creating the update and install procedure for the new release, but also some of it is inherent in our academic setting: we must not make major changes to workstation software while an academic term is in progress.

Our security model gains much clarity and agility by assuming serial reuse. Several problems still inhere. First, dialup servers are used by many users at once, and must be carefully managed. We have no general way to deal with such machines, and they require specialized support of various kinds. Second, we have no way to easily handle long-running interactive jobs. And finally, users cannot be fully confident that the software on a given workstation has not been compromised in some security-related way.

All told, however, the Athena system has borne the years very well. We manage a very large heterogeneous network with a fairly small staff of programmers and developers. It is hard to imagine how we could do so without the tools described in this paper, or their equivalents.

Author Information

Thomas Bushnell, BSG works for MIT Information Systems as a systems programmer in the Athena team. Previously he worked for over seven years as a programmer and software architect for the Free Software Foundation, writing the GNU Hurd and helping with several other projects. He studies philosophy, classics, and many other things, and can be reached via email at tb@mit.edu.

Karl Ramm is a systems programmer for MIT Information Systems, where he helps keep Athena from falling over. He can be reached via U.S. Mail at Room E40-342C, Massachusetts Institute of Technology; 1 Amherst St.; Cambridge MA 02139, while he can be reached electronically at kcr@mit.edu. He has written five different mail reading programs and has never written a windowing system.

References

- [Kerberos] Steiner, Neuman, Schiller, "Kerberos: An Authentication Service for Open Network Systems," *USENIX Technical Conference*, Dallas, Texas, Winter 1988. <<ftp://athena-dist.mit.edu/pub/ATHENA/usenix/kerberos.PS>>
- [Hesiod] Dyer, Stephen P., "The Hesiod Name Server," *USENIX Technical Conference*, Dallas, Texas, Winter 1988. <<ftp://athena-dist.mit.edu/pub/ATHENA/usenix/hesiod.PS>>
- [ATPB] Lerman and Saltzer, "Section B: Technical Objectives and Requirements," *Project Athena Technical Plan*, M.I.T. Project Athena, Cambridge, Massachusetts, July 23, 1987.
- [ATPC] Balkovich, Parmaless, and Saltzer, "Section C: Project Athena's Model of Computation," *Project Athena Technical Plan*, M.I.T. Project Athena, Cambridge, Massachusetts, September 16, 1985.
- [ATPD] Saltzer, J. H., "Section D: Evolution to the Athena Workstation Model: An Overview of the Development Plan", *Project Athena Technical Plan*, M.I.T. Project Athena, Cambridge, Massachusetts, March 6, 1986.
- [RelEng] Davis, Don, "Project Athena's Release Engineering Tricks," M.I.T. Project Athena, Cambridge, Massachusetts. <<ftp://athena-dist.mit.edu/pub/ATHENA/usenix/rlseng.PS>>
- [Track] Nachbar, Daniel, "When Network File Systems Aren't Enough: Automatic File Distribution Revisited.," *USENIX Technical Conference*, Summer, 1986.

Bootstrapping an Infrastructure

Steve Traugott – Sterling Software and NASA Ames Research Center
Joel Huddleston – Level 3 Communications

ABSTRACT

When deploying and administering systems infrastructures it is still common to think in terms of individual machines rather than view an entire infrastructure as a combined whole. This standard practice creates many problems, including labor-intensive administration, high cost of ownership, and limited generally available knowledge or code usable for administering large infrastructures.

The model we describe treats an infrastructure as a single large distributed virtual machine. We found that this model allowed us to approach the problems of large infrastructures more effectively. This model was developed during the course of four years of mission-critical rollouts and administration of global financial trading floors. The typical infrastructure size was 300-1000 machines, but the principles apply equally as well to much smaller environments. Added together these infrastructures totaled about 15,000 hosts. Further refinements have been added since then, based on experiences at NASA Ames.

The methodologies described here use UNIX and its variants as the example operating system. We have found that the principles apply equally well, and are as sorely needed, in managing infrastructures based on other operating systems.

This paper is a living document: Revisions and additions are expected and are available at www.infrastructures.org. We also maintain a mailing list for discussion of infrastructure design and implementation issues – details are available on the web site.

Introduction

There is relatively little prior art in print which addresses the problems of large infrastructures in any holistic sense. Thanks to the work of many dedicated people we now see extensive coverage of individual tools, techniques, and policies [nemeth] [frisch] [stern] [dns] [evard] [limoncelli] [anderson]. But it is difficult in practice to find a “how to put it all together” treatment which addresses groups of machines larger than a few dozen.

Since we could find little prior art, we set out to create it. Over the course of four years of deploying, reworking, and administering large mission-critical infrastructures, we developed a certain methodology and toolset. This development enabled thinking of an entire infrastructure as one large “virtual machine,” rather than as a collection of individual hosts. This change of perspective, and the decisions it invoked, made a world of difference in cost and ease of administration.

If an infrastructure is a virtual machine, then creating or reworking an infrastructure can be thought of as booting or rebooting that virtual machine. The concept of a boot sequence is a familiar thought pattern for sysadmins, and we found it to be a relatively easy one to adapt for this purpose.

We recognize that there really is no “standard” way to assemble or manage large infrastructures of UNIX machines. While the components that make up

a typical infrastructure are generally well-known, professional infrastructure architects tend to use those components in radically different ways to accomplish the same ends. In the process, we usually write a great deal of code to glue those components together, duplicating each others’ work in incompatible ways.

Because infrastructures are usually ad hoc, setting up a new infrastructure or attempting to harness an existing unruly infrastructure can be bewildering for new sysadmins. The sequence of steps needed to develop a comprehensive infrastructure is relatively straightforward, but the discovery of that sequence can be time-consuming and fraught with error. Moreover, mistakes made in the early stages of setup or migration can be difficult to remove for the lifetime of the infrastructure.

We will discuss the sequence that we developed and offer a brief glimpse into a few of the many tools and techniques this perspective generated. If nothing else, we hope to provide a lightning rod for future discussion. We operate a web site (www.infrastructures.org) and mailing list for collaborative evolution of infrastructure designs. Many of the details missing from this paper should show up on the web site.

In our search for answers, we were heavily influenced by the MIT Athena project [athena], the OSF Distributed Computing Environment [dce], and by work done at Carnegie Mellon University [sup] [afs] and the National Institute of Standards and Technology [depot].

Infrastructure Thinking

We found that the single most useful thing a would-be infrastructure architect can do is develop a certain mindset: A good infrastructure, whether departmental, divisional, or enterprise-wide, is a single loosely-coupled virtual machine, with hundreds or thousands of hard drives and CPU's. It is there to provide a substrate for the enterprise to do its job. If it doesn't do that, then it costs the enterprise unnecessary resources compared to the benefit it provides. This extra cost is often reflected in the attitude the enterprise holds towards its systems administration staff. Providing capable, reliable infrastructures which grant easy access to applications makes users happier and tends to raise the sysadmin's quality of life. See the *Cost of Ownership* section.

This philosophy overlaps but differs from the "dataless client" philosophy in a subtle but important way: It discourages but does not preclude putting unique data on client hard disks, and provides ways to manage it if you do. See the *Network File Servers*, *Client File Access*, and *Client Application Management* sections.

The "virtual machine" concept simplified how we maintained individual hosts. Upon adopting this mindset, it immediately became clear that all nodes in a "virtual machine" infrastructure needed to be generic, each providing a commodity resource to the infrastructure. It became a relatively simple operation to add, delete, or replace any node. See the *Host Install Tools* section.

Likewise, catastrophic loss of any single node caused trivial impact to users. Catastrophic loss of an entire infrastructure was as easy to recover from as the loss of a single traditionally-maintained machine. See the *Disaster Recovery* section.

When we logged into a "virtual machine," we expected to use the same userid and password no matter which node we logged into. Once authenticated, we were able to travel with impunity throughout the "machine" across other nodes without obstruction. This was true whether those nodes sat on a desktop or in a server room. In practice, this idea can be modified to include the idea of "realms" of security which define who can access certain protected areas of the virtual machine. You might want to implement a policy that disallows ordinary user logins on nodes of class "NFS server," for instance. Note that this approach is markedly different from explicitly giving users logins on each individual machine. By classing machines, you ensure that when a new machine is added to a class, the correct users will already be able to log into it. See the *Authentication Servers* section.

Adds, moves, and changes consume a great deal of time in a traditional infrastructure because people's workstations have to be physically moved when the people move. Computing itself is enabling

organizations to become more dynamic – meaning reorgs are becoming more prevalent. This makes free seating critical in modern infrastructures.

In a "virtual machine" infrastructure made up of commodity nodes, only the people need to move; they log off of their old workstation, walk over to their new desk, sit down, log in, and keep working. They see the same data and binaries, accessed via the same pathnames and directory structure, no matter which node they log into. This is well within the capabilities of modern automounters and NFS, particularly if you are willing to add some Perl glue and symbolic link farms. See the *Client File Access* and *Client Application Management* sections.

Traditionally, installing an application or patch means visiting each machine physically or over the net to install that package. In a "virtual machine" infrastructure, you "install" the package once by dropping it into a central repository and letting it propagate out from there to all of the hard disks. See the *File Replication Servers* and *Client OS Update Methods* sections.

The Infrastructure Bootstrap Sequence

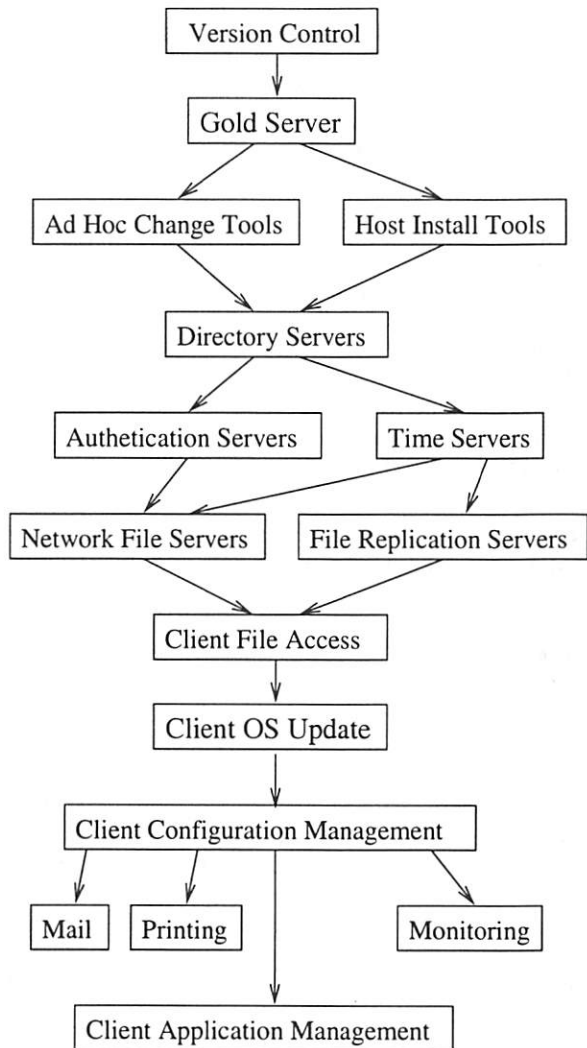
A certain sequence of events needs to occur while creating a virtual machine infrastructure. Most of these events are dependent on earlier events in the sequence. Mistakes in the sequence can cause non-obvious problems, and delaying an event usually causes a great deal of extra work to compensate for the missing functionality. These relationships are often not readily apparent in the "heat of the moment" of a rollout.

We found that keeping this sequence in mind was invaluable whether creating a new infrastructure from vanilla machines fresh out of the box, or migrating existing machines already in place into a more coherent infrastructure.

If you are creating a new infrastructure from scratch and do not have to migrate existing machines into it, then you can pretty much follow the bootstrap sequence as outlined below. If you have existing machines which need to be migrated, see the *Migrating From an Existing Infrastructure* section.

As mentioned earlier, the following model was developed during the course of four years of mission-critical rollouts and administration of global financial trading floors. The typical infrastructure size was 300-1000 machines, totaling about 15,000 hosts. Nothing precludes you from using this model in much smaller environments – we've used it for as few as three machines. This list was our bible and roadmap – while incomplete and possibly not in optimum order, it served its purpose. See Figure 1 for an idea of how these steps fit together.

The following sections describe these steps in more detail.



1. **Version Control** – CVS, track who made changes, backout
2. **Gold Server** – only require changes in one place
3. **Host Install Tools** – install hosts without human intervention
4. **Ad Hoc Change Tools** – ‘expect’, to recover from early or big problems
5. **Directory Servers** – DNS, NIS, LDAP
6. **Authetication Servers** – NIS, Kerberos
7. **Time Synchronization** – NTP
8. **Network File Servers** – NFS, AFS, SMB
9. **File Replication Servers** – SUP
10. **Client File Access** – automount, AMD, autolink
11. **Client OS Update** – rc.config, configure, make, cfengine
12. **Client Configuration Management** – cfengine, SUP, CVSup
13. **Client Application Management** – autosup, autolink
14. **Mail** – SMTP
15. **Printing** – Linux/SMB to serve both NT and UNIX
16. **Monitoring** – syslogd, paging

Figure 1: Infrastructure Bootstrap Sequence.

Step 1: Version Control

Prerequisites: none.

Lack of version control over your infrastructure leads to eventual confusion. We used version control for tracking OS configuration files, OS and application binaries and source code, and tools and administrative scripts. We managed independent evolution of several infrastructures, and were able to do rollbacks or rebuilds of damaged servers and other components.

It may seem strange to start with version control. Many sysadmins go through their entire careers without it. But infrastructure building is fundamentally a development process, and a great deal of shell, Perl, and other code tends to get generated. We found that once we got good at “doing infrastructures,” and started getting more work thrown at us, we had several distinct infrastructures at various stages of development at any given time. These infrastructures were often in different countries, and always varied slightly from each other. Managing code threatened to become a nightmare.

We found that CVS helped immensely in managing many different versions and branches of administrative code trees [cvs]. It took some careful thought and some tool building to be able to cram O/S configuration files and administrative code into a CVS repository and make it come back out okay on all the right machines. It was worth the effort. In later iterations, we began migrating the hundreds of megabytes of vendor-supplied O/S code itself into the CVS repositories, with some success. The latest version of CVS (1.10) has additional features which would have made this much easier, such as managing symbolic links natively.

Since all of our code was mastered from the CVS repository, we could actually destroy entire server farms and rebuild them with relative impunity during the course of development, moves, or disaster recovery. This also made it much easier to roll back from undesired changes.

In short, based on our experience, we’d strongly advise setting up and using CVS and associated tools

as the first step in an infrastructure development program.

We tried various vendor-supplied version control tools – everyone had their favorites. While many of these seemed to offer better features than CVS, none of them turned out to be flexible, robust, or WAN-centric enough to manage operating system code in-place on live machines scattered all over the world. Because of its Internet heritage and optimized use on far-flung projects [samba] [bsd], CVS was almost perfect for this. Where it wasn't, we were able to get under the hood and get what we needed in a way that we would never have been able to with a proprietary tool.

Step 2: Gold Server

Prerequisites: version control.

We used CVS to manage only one machine in each distinct infrastructure – the “gold server.” Changes to any other machine in the infrastructure had to propagate out from the gold server. This allowed us to make our changes reproducible, recoverable, traceable, and able to be ported and integrated into our other infrastructures. The results were rewarding: We were able to make a true migration from “systems administrators” to “infrastructure engineers.” We learned to abhor fixing the same thing twice, and got to spend our time working out fun, complex puzzles of infrastructure design (and then going home earlier).

We can't stress enough the fact that our gold server was passive. Understanding this concept is key to understanding our model. The gold server served files via NFS, SUP [sup], and CVS [cvs], and that's all. Client machines were responsible for periodically contacting the gold server to obtain updates. Neither the gold server nor any other mechanism ever “pushed” changes to clients asynchronously. See the *Push vs. Pull* section.

The gold server was an interesting machine; it usually was not part of the infrastructure, was usually the only one-off in the whole infrastructure, was not mission-critical in the sense that work stopped if it went down, but nevertheless the entire infrastructure grew from and was maintained by that one machine. It was the network install server, the patch server, the management and monitoring server, and was often the most protected machine from a security standpoint.

We developed a rule that worked very well in practice and saved us a lot of heartache: “Never log into a machine to change anything on it. Always make the change on the gold server and let the change propagate out.”

We managed the gold server by maintaining it as an ordinary CVS sandbox, and then used SUP to replicate changes to client disks. It might make more sense today to use CVSup [polstra]. (See the *File Replication* section.)

We used one gold server for an entire infrastructure; this meant binaries had to be built on other

platforms and transferred to the gold server's NFS or replication server trees. Other infrastructures we've seen use a different gold server for every hardware/OS combination.

Step 3: Host Install Tools

Prerequisites: Gold Server.

We managed all of our desktop machines identically, and we managed our server machines the same way we managed our desktop machines. We usually used the vendor-supplied OS install tool to place the initial disk image on new machines. The install methods we used, whether vendor-supplied or homebuilt, were usually automatic and unattended. Install images, patches, management scripts, and configuration files were always served from the gold server.

We managed desktops and servers together because it's much simpler that way. We generally found no need for separate install images, management methodologies, or backup paradigms for the two. Likewise, we had no need nor desire for separate “workstation” and “server” sysadmin groups, and the one instance this was thrust upon us for political reasons was an unqualified disaster.

The only difference between an NFS server and a user's desktop machine usually lay in whether it had external disks attached and had anything listed in /etc/exports. If more NFS daemons were needed, or a kernel tunable needed to be tweaked, then that was the job of our configuration scripts to provide for at reboot, after the machine was installed. This boot-time configuration was done on a reproducible basis, keyed by host name or class. (See the *Client OS Update* and *Client Configuration Management* sections.)

We did not want to be in the business of manually editing /etc/* on every NFS server, let alone every machine – it's boring and there are better things for humans to do. Besides, nobody ever remembers all of those custom tweaks when the boot disk dies on a major NFS server. Database, NIS, DNS, and other servers are all only variations on this theme.

Ideally, the install server is the same machine as the gold server. For very large infrastructures, we had to set up distinct install servers to handle the load of a few hundred clients all requesting new installs at or near the same time.

We usually used the most vanilla O/S image we could, often straight off the vendor CD, with no patches installed and only two or three executables added. We then added a hook in /etc/rc.local or similar to contact the gold server on first boot.

The method we used to get the image onto the target hard disk was always via the network, and we preferred the vendor-supplied network install tool, if any. For SunOS we wrote our own. For one of our infrastructures we had a huge debate over whether to use an existing in-house tool for Solaris, or whether to use Jumpstart. We ended up using both, plus a simple

'dd' via 'rsh' when neither was available. This was not a satisfactory outcome. The various tools inevitably generated slightly different images and made subsequent management more difficult. We also got too aggressive and forgot our rule about "no patches," and allowed not only patches but entire applications and massive configuration changes to be applied during install on a per-host basis, using our in-house tool. This, too, was unsatisfactory from a management standpoint; the variations in configuration required a guru to sort out.

Using absolutely identical images for all machines of a given hardware architecture works better for some O/S's than for others; it worked marvelously for AIX, for instance, since the AIX kernel is never rebuilt and all RS/6000 hardware variants use the same kernel. On SunOS and Solaris we simply had to take the different processor architectures into account when classing machines, and the image install tool had to include kernel rebuilds if tunables were mandatory.

It's important to note that our install tools generally required only that a new client be plugged in, turned on, and left unattended. The result was that a couple of people were able to power up an entire floor of hundreds of machines at the same time and then go to dinner while the machines installed themselves. This magic was usually courtesy of bootp entries on the install server pointing to diskless boot images which had an "install me" command of some sort in the NFS-mounted /etc/rc.local. This would format the client hard drive, 'dd' or 'cpio' the correct filesystems onto it, set the hostname, domain name, and any other unique attributes, and then reboot from the hard disk.

Step 4: Ad Hoc Change Tools

Prerequisites: installed hosts in broken state running rshd, sshd, or telnetd.

Push-based ad hoc change tools such as r-commands and expect scripts are detrimental to use on a regular basis. They generally cause the machines in your infrastructure to drift relative to each other. This makes your infrastructure more expensive to maintain and makes large-scale disaster recovery infeasible. There are few instances where these tools are appropriate to use at all.

Most sysadmins are far too familiar with ad hoc change, using rsh, rcp, and rdist. We briefly debated naming this paper "rdist is not your friend." If we ever write a book about enterprise infrastructures, that will be the title of one of the chapters. Many will argue that using ad hoc tools to administer a small number of machines is still the cheapest and most efficient method. We disagree. Few small infrastructures stay small. Ad hoc tools don't scale. The habits and scripts you develop based on ad hoc tools will work against you every time you are presented with a larger problem to solve.

We found that the routine use of ad hoc change tools on a functioning infrastructure was the strongest contributor towards high total cost of ownership (TCO). This seemed to be true of every operating system we encountered, including non-UNIX operating systems such as Windows NT and MacOS.

Most of the cost of desktop ownership is labor [gartner], and using ad hoc change tools increases entropy in an infrastructure, requiring proportionally increased labor. If the increased labor is applied using ad hoc tools, this increases entropy further, and so on – it's a positive-feedback cycle. Carry on like this for a short time and all of your machines will soon be unique even if they started out identical. This makes development, deployment, and maintenance of applications and administrative code extremely difficult (and expensive).

Ordinarily, any use that we did make of ad hoc tools was simply to force a machine to contact the gold server, so any changes which did take place were still under the gold server's control.

After you have done the initial image install on 300 clients and they reboot, you often find they all have some critical piece missing that prevents them from contacting the gold server. You can fix the problem on the install image and re-install the machines again, but time constraints may prevent you from doing that. In this case, you may need to apply ad hoc tools.

For instance, we usually used entries in our machines' rc.local or crontab, calling one or two executables in /usr/local/bin, to trigger a contact with the gold server (via NFS or SUP) on every boot. If any of this was broken we had to have an ad hoc way to fix it or the machine would never get updates.

Since the "critical piece missing" on newly installed hosts could be something like /.rhosts or hosts.equiv, that means rcp, rsh, or ssh can't be counted on. For us, that meant 'expect' [libes] was the best tool.

We developed an expect script called 'rabbit' [rabbit] which allowed us to execute arbitrary commands on an ad hoc basis on a large number of machines. It worked by logging into each of them as an appropriate user, ftp'ing a small script into /tmp, and executing it automatically.

Rabbit was also useful for triggering a pull from the gold server when we needed to propagate a change right away to hundreds of machines. Without this, we might have to wait up to an hour for a crontab entry on all the client machines to trigger the pull instead.

Step 5: Directory Servers

Prerequisites: Host Install Tools.

You'll need to provide your client machines with hostname resolution, UID and GID mappings, automount maps, and possibly other items of data that are generally read-only (this does not include

authentication – see the *Authentication Servers* section). The servers you use for these functions should be part of your infrastructure rather than standalone. The master copies of the data they serve will need to be backed up somewhere easily accessible.

You'll probably want to use DNS for hostnames, and either use NIS or file replication for UID, GID, and automounter mapping.

Here are some things to consider while choosing directory services:

- Is the protocol common on every machine you are likely to use? Some protocols, most notably NIS+, have very limited availability.
- Does it work on every machine you are likely to use? A poor implementation of NIS often forced us to use file replication instead.
- Is it unnecessarily complicated? A full-featured database with roll-back and checkpoints to perform IP service name to number mapping is probably overkill.
- How much will you have to pay to train new administrators? An esoteric, in-house system may solve the problem, but what happens when the admin who wrote and understands it leaves?
- Is it ready for prime-time? We used one product for a while for authentication services that we wanted to abandon because we kept hearing "Oh, that is available in the next release."

DNS, NIS and the file replication tools described in the following sections eventually all became necessary components of most of our infrastructures. DNS provided hostname to IP address mapping, as it was easy to implement and allowed subdomain admins to maintain their hosts without appealing to a corporate registry. DNS is also the standard for the Internet – a fact often lost in the depths of some corporate environments. NIS provided only the authentication mechanism, as described in the next section. NIS may not be the best choice, and we often wanted to replace it because of the adverse affects NIS has on a host when the NIS servers are all unreachable.

We wanted our machines to be able to boot with no network present. This dictated that each of our clients be a NIS slave. Pulling the maps down on an hourly or six-minute cycle and keeping hundreds of 'ypserv' daemons sane required writing a good deal of management code which ran on each client. Other infrastructures we've seen also make all clients caching DNS servers.

We recommend that directory server hosts not be unique, standalone, hand-built machines. Use your host install tools to build and configure them in a repeatable way, so they can be easily maintained and your most junior sysadmin can quickly replace them when they fail. We found that it's easy to go overboard with this though: It's important to recognize the difference between mastering the server and mastering the data it's serving. Mastering the directory database

contents from the gold server generally guarantees problems unless you always use the gold server (and the same mastering mechanism) to make modifications to the database, or if you enforce periodic and frequent dumps to the gold server from the live database. Other methods of managing native directory data we've seen include cases such as mastering DNS data from a SQL database.

We used hostname aliases in DNS, and in our scripts and configuration files, to denote which hosts were offering which services. This way, we wouldn't have to edit scripts when a service moved from one host to another. For example, we had CNAMEs of 'sup' for the SUP server, 'gold' for the gold server, and 'cvs' for the CVS repository server, even though these might all be the same machine.

Step 6: Authentication Servers

Prerequisites: Directory Servers, so clients can find user info and authentication servers.

We wanted a single point of authentication for our users. We used NIS. The NIS domain name was always the same as the DNS domain name. It's possible we could have treed out NIS domains which were subsets of the DNS domain, but we didn't think we needed to.

We'd like to clarify how we differentiate between a simple directory service and an authentication service: A directory service supplies information through a one-way trust relationship – the client trusts the server to give accurate information. This trust typically comes from the fact that a local configuration file (*resolv.conf*, *ypservers*) tells the client which server to contact. This is part of an authentication service, but there is a fine distinction.

An authentication service supplies an interaction that develops a two-way trust. The client uses the service to prove itself trustworthy. The UNIX login process provides a good example of this interaction. The client (in this case, a person) enters a text string, the password. This is compared to a trusted value by the server (the UNIX host.) If they do not match, no trust is developed. Login is denied. If they do match, the user is rewarded with control of a process operating under their name and running their shell. The whole point of an authentication service is that it allows the client to prove itself to be trustworthy, or at least to prove itself to be the same nefarious character it claims.

NIS, NIS+, Kerberos, and a raft of commercial products can be used to provide authentication services. We went through endless gyrations trying to find the "perfect" authentication service. We kept on ending up back at NIS, not because we liked it so much as because it was there.

It's useful to note that there are really only four elements to a user's account in UNIX – the encrypted password, the other info contained in */etc/passwd*

(such as UID), the info contained in `/etc/group`, and the contents of the home directory. To make a user you have to create all of these. Likewise, to delete a user you have to delete all of these.

Of these four elements, the encrypted password is the most difficult to manage. The UID and GID mappings found in `/etc/passwd` and `/etc/group` can easily be distributed to clients via file replication (see the *File Replication* section). The home directory is usually best served via NFS and automounter (see the *Client File Access* section).

In shadow password implementations, the encrypted password is located in a separate database on the local host. In implementations such as NIS and Kerberos, the encrypted password and the mechanisms used to authenticate against it are moved totally off the local host onto a server machine.

We wanted to develop a single point of authentication for our users. This meant either replicating the same `/etc/passwd`, `/etc/group`, and `/etc/shadow` to all machines and requiring users to always change their password on a master machine, or using NIS, or installing something like Kerberos, or putting together an in-house solution.

It's interesting to note that even if we had used Kerberos we still would have needed to replicate `/etc/passwd` and `/etc/group`; Kerberos does not provide the information contained in these files.

What we usually ended up doing was using NIS and replicating `/etc/passwd` and `/etc/group` with minimal contents. This way we were able to overlay any local changes made to the files; we didn't want local users and groups proliferating.

In keeping with the "virtual machine" philosophy, we always retained a one-to-one mapping between the borders of the DNS and NIS domains. The NIS domain name was always the same as the DNS domain name. This gave us no leeway in terms of splitting our "virtual machines" up into security realms, but we found that we didn't want to; this kept things simple.

If you do want to split things up, you might try subclassing machines into different DNS subdomains, and then either use NIS+ subdomains, hack the way NIS generates and distributes its maps to create a subdomain-like behavior, or use different Kerberos realms in these DNS subdomains. Either way, these DNS subdomains would be children of a single parent DNS domain, all of which together would be the virtual machine, with only one gold server to tie them all together.

A note about username strings and keeping users happy: In today's wired world, people tend to have many login accounts, at home, at work, and with universities and professional organizations. It's helpful and will gain you many points if you allow users to pick their own login name, so they can keep all of

their worlds synchronized. You don't have to look at and type that name every day – they do, over and over. They will think of you every time. You want that thought to be a positive one.

Step 7: Time Synchronization

Prerequisites: Directory Servers, so clients can find the time servers.

Without good file timestamps, backups don't work correctly and state engines such as 'make' get confused. It's important not to delay implementing time synchronization, probably by using NTP.

Many types of applications need accurate time too, including scientific, production control, and financial. It's possible for a financial trader to lose hundreds of thousands of dollars if he refers to a workstation clock which is set wrong. A \$200 radio clock and NTP can be a wise investment.

Shy away from any tool which periodically pops machines into the correct time. This is the solution implemented on several PC based systems. They get their time when they connect to the server and then never update again. It works for these systems because they do not traditionally stay up for long periods of time. However, when designing for the infrastructure, it helps to think that every system will be up 24x7 for months, or even years, between reboots. Even if you put a "time popper" program in crontab, bizarre application behavior can still result if it resets the clock backwards a few seconds every night.

Eventually, you will implement NTP [ntp]. It is only a matter of time. NTP has become a standard for time services in the same way that DNS has become a standard for name services. The global NTP stratum hierarchy is rooted at the atomic clocks at the NIST and Woods Hole. You can't get much more authoritative. And NTP drifts clocks by slowing them down or speeding them up – no "popping."

If your network is connected to the Internet, your ISP may provide a good NTP time source.

Even if you need to run an isolated set of internal time servers, and sync to the outside world by radio clock or wristwatch, NTP is still the better choice because of the tools and generic knowledge pool available. But you may want to have only one stratum 1 server in this case; see the NTP docs for an explanation. You should also prefer a radio clock over the wristwatch method – see the trader example above.

For large infrastructures spread over many sites, you will want to pick two or three locations for your highest stratum NTP servers. Let these feed regional or local servers and then broadcast to the bottom tier.

Step 8: Network File Servers

Prerequisites: Directory Servers, so clients can find the file servers, Authentication Servers, so clients can verify users for file access, Time Servers, so clients agree on file timestamps.

We kept our file servers as generic and identical to each other as possible. There was little if any difference between a client and server install image. This enabled simple recovery. We generally used external hardware RAID arrays on our file servers. We often used High-Availability NFS servers [blide]. We preferred Samba [samba] when serving the same file shares to both UNIX and Windows NT clients. We were never happy with any “corporate” backup solutions – the only solution we’ve ever come close to being satisfied with on a regular basis is Amanda [amanda].

The networked UNIX filesystem has been reinvented a few times. In addition to NFS we have AFS, DFS, and CacheFS, to name a few. Of these, only NFS was available on all of our client platforms, so for us it was still the best choice. We might have been able to use AFS in most cases, but the expense, complexity, and unusual permissions structure of AFS were obstacles to its implementation. And if AFS is complex, then DFS is even more so.

One interesting network filesystem is Coda [coda]. Currently under development but already publicly available, this non-proprietary caching filesystem is freely available, and already ported to many operating systems, including Linux. It supports disconnected operation, replicated servers, and Kerberos authentication. These features when added together may make it worth the complexity of implementation.

An open-source implementation of CacheFS would also be good.

As mentioned before, we kept the disk image differences between a desktop client and an NFS server to a minimum. With few exceptions, the only differences between a desktop and server machine were whether it had external disks attached and the speed and number of processors. This made maintenance easy, and it also made disaster recovery simple.

Step 9: File Replication Servers

Prerequisites: Directory Servers, Time Synchronization.

Some configuration files will always have to be maintained on the client’s local hard drive. These include much of `/etc/*`, and in our case, the entire `/usr/local` tree. How much you keep on your local disk is largely determined by how autonomous you want your machines to be. We periodically replicated changed files from the gold server to the local hard disks.

We needed a fast, incremental and mature file replication tool. We chose Carnegie Mellon’s SUP (Software Upgrade Protocol) [sup]. We would have preferred a flexible, portable, open-source caching file system, but since none were available we opted for this “poor man’s caching” instead. It worked very well.

Aside from the advantage of incremental updates, SUP offered a strict “pull” methodology. The client, not the server, chose the point in time when it would be updated. (See the *Push vs. Pull* section.)

Using this mechanism, we were able to synchronize the files in `/etc` on every client every six minutes, and the contents of `/usr/local` every hour. (This on a trading floor with over 800 clients.)

We also replicated selected applications from the NFS servers to the client hard disks. (See the *Client Application Management* section.)

We used SUP to replicate most of the files that NIS normally manages, like `/etc/services` and the automounter maps. We only used NIS to manage authentication – the `passwd` map.

A more recent development, familiar to many open source developers and users, is CVSup [polstra]. With ordinary SUP, we had to do a ‘cvs update’ in the replication source tree on the gold server to check the latest changes out of the CVS repository. We then used SUP jobs in `crontab` to pull the changes from there down to the client. Today it may make more sense to skip the intermediary step, and instead use CVSup to pull files and deltas directly from the CVS repository into the live locations on the client hard disks.

Step 10: Client File Access

Prerequisites: Network File Servers, File Replication Servers.

We wanted a uniform filesystem namespace across our entire virtual machine. We were able to move data from server to server without changing pathnames on the clients. We also were able to move binaries from servers to client disks or back without changing the pathnames the binaries were executed from. We used automounters and symlink farms extensively. We would have liked to see good open-source caching filesystems.

CacheFS was ruled out as a general solution because of its limited heterogeneity. We might have been able to use CacheFS on those clients that offered it, but that would have required significantly different management code on those clients, and time constraints prevented us from developing this further.

In keeping with the virtual machine concept, it is important that every process on every host see the exact same file namespace. This allows applications and users to always find their data and home directories in the same place, regardless of which host they’re on. Likewise, users will always be able to find their applications at the same pathname regardless of hardware platform.

If some clients have an application installed locally, and others access the same application from a file server, they both should “see” the application in the same place in the directory tree of the virtual machine. We used symbolic link “farms” in the `/apps` directory that pointed to either `/local/apps` or

/remote/apps, depending on whether the application was installed locally or remotely. The /local/apps filesystem was on the client hard disk, while /remote/apps was composed of automounted filesystems from NFS servers. [mott]

One tiny clue to better understanding of our model is this: the directories served by an NFS server were always served from /local/apps on the server itself. Also, /usr/local was always a symlink to /local. One of our tenets was that all data unique to a machine and not part of the OS be stored in /local. This way we could usually grab all of the critical and irreplaceable uniqueness of a machine by grabbing the contents of /local. (OS-related uniqueness goes in /var, as always.)

The automounter has some pitfalls: Indirect mounts are more flexible than direct mounts, and are usually less buggy. If a vendor's application insists that it must live at /usr/appname and you want to keep that application on a central server, resist the temptation to simply mount or direct automount the directory to /usr/appname. UNIX provides the symbolic link to solve this problem. Point the /usr/appname symlink at an indirect mapped /remote/apps (or similar) directory. Similarly, a common data directory (perhaps, /data) managed by an indirect map could be defined for any shared data that must be writable by the clients.

Another serious danger is the use of /net. Automounters have the ability to make all exports from a server appear at /net/servername or something similar. This is very handy for trouble-shooting and quick maintenance hacks. It can, however, put an oppressive load on the server if the server is exporting a large number of filesystems – cd'ing to /net/scotty will generate a mount request for all of scotty's filesystems at once. Worse, it reduces the flexibility of your infrastructure, because host names become a part of the file name. This prevents you from moving a file to a new server without changing every script and configuration file which refers to it.

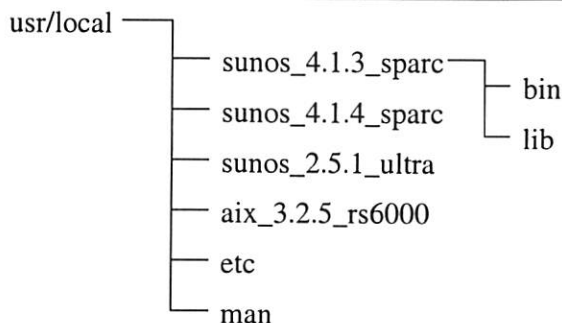


Figure 2: Example of a Heterogeneous /usr/local SUP Server Tree.

It was difficult for us to come up with a heterogeneous filesystem naming convention. We finally settled on installing a script (/usr/local/bin/platform) on every machine which, when run, spit out a formatted version of the output of 'uname -a'. The naming

convention we used looked something like 'sunos_4.1.4_sparc', 'sunos_5.1.5_ultra', and 'aix_3.2.5_rs6000'. This script was called from everywhere; automounters, boot scripts, application startup scripts, and the makefile described below. We used this platform string in many places, including heterogeneous directory paths. See Figure 2. We made 'platform' a script, not a simple data file, to guard against the possibility that out-of-date information would cause errors.

Step 11: Client O/S Update

Prerequisites: Network File Servers, File Replication Servers.

Vendors are waking up to the need for decent, large scale operating systems upgrade tools. Unfortunately, due to the "value added" nature of such tools, and the lack of published standards, the various vendors are not sharing or cooperating with one another. It is risky to use these tools even if you think you will always have only one vendor to deal with. In today's business world of mergers and reorgs, single vendor networks become a hodge-podge of conflicting heterogeneous networks overnight.

We started our work on a homogeneous network of systems. Eventually we added a second, and then a third OS to that network. We took about five months adding the second OS. When the third came along, we found that adding it to our network was a simple matter of porting the tools – it took about a week. Our primary tool was a collection of scripts and binaries that we called Hostkeeper.

Hostkeeper depended on two basic mechanisms; boot time configuration and ongoing maintenance. At boot, the Hostkeeper client contacted the gold server to determine whether it had the latest patches and upgrades applied to its operating system image. This contact was via an NFS filesystem (/is/conf) mounted from the gold server.

We used 'make' for our state engine. Each client always ran 'make' on every reboot. Each OS/hardware platform had a makefile associated with it (/is/conf/bin/Makefile.{platform}). The targets in the makefile were tags that represented either our own internal revision levels or patches that made up the revision levels. We borrowed a term from the aerospace industry – "block 00" was a vanilla machine, "block 10" was with the first layer of patches installed, and so on. The Makefiles looked something like Listing 1. Note the 'touch' commands at the end of each patch stanza; this prevented 'make' from running the same stanza on the same machine ever again. (We ran 'make' in a local directory where these timestamp files were stored on each machine.)

We had mechanisms that allowed us to manage custom patches and configuration changes on selected machines. These were usually driven by environment variables set in /etc/environment or the equivalent.

The time required to write and debug a patch script and add it to the makefile was minimal compared to the time it would have taken to apply the same patch to over 200 clients by hand, then to all new machines after that. Even simple changes, such as configuring a client to use a multi-headed display, were scripted. This strict discipline allowed us to exactly recreate a machine in case of disaster.

For operating systems which provided a patch mechanism like 'pkgadd', these scripts were easy to write. For others we had our own methods. These days we would probably use RPM for the latter [rpm].

You may recognize many of the functions of 'cfengine' in the above description [burgess]. At the time we started on this project, 'cfengine' was in its early stages of development, though we were still tempted to use it. If we had this to do again it's likely 'cfengine' would have supplanted 'make'.

One tool that bears closer scrutiny is Sun Microsystems' Autoclient. The Autoclient model can best be described as a dataless client whose local files are a cached mirror of the server. The basic strategy of Autoclient is to provide the client with a local disk drive to hold the operating system, and to refresh that operating system (using Sun's CacheFS feature) from a central server. This is a big improvement over the old diskless client offering from Sun, which overloaded servers and networks with NFS traffic.

One downside of Autoclient is its dependence on Sun's proprietary CacheFS mechanism; another is its scalability. Eventually, the number of clients will exceed that which one server can support. This means adding a second server, then a third, and then the problem becomes one of keeping the servers in sync. Essentially, Autoclient does not solve the problem of system synchronization; it delays it. However, this delay may be exactly what the system administrator needs to get a grip on a chaotic infrastructure.

```
block00: localize
block10: block00 14235-43 xdm_fix01
14235-43 xdm_fix01:
    /is/conf/patches/${PLATFORM}/${@}/install_patch
    touch ${@}
localize:
    /is/conf/bin/localize
    touch ${@}
```

Listing 1: Hostkeeper makefile example.

```
root:all:1 2 * * * [-x /usr/sbin/rtc] && /usr/sbin/rtc -c > /dev/null 2>&1
root:all:0 2 * * 0,4 /etc/cron.d/logchecker
root:all:5 4 * * 6 /usr/lib/newsyslog
root:scotty:0 4 * * * find . -fstype nfs -prune -o -print >/var/spool/lSR
stevegt:skywalker:10 0-7,19-23 * * * /etc/reset_tiv
[...]
```

Listing 2: Crontab.master file.

Step 12: Client Configuration Management

Prerequisites: Network File Servers, File Replication Servers.

In a nutshell, client configuration is localization. This includes everything that makes a host unique, or that makes a host a participant of a particular group or domain. For example, hostname and IP addresses must be different on every host. The contents of /etc/resolv.conf should be similar, if not identical, on hosts that occupy the same subnet. Automount maps which deliver users' home directories must be the same for every host in an authentication domain. The entries in client crontabs need to be mastered from the gold server.

Fortunately, if you have followed the roadmap above, most of this will fall into place nicely. If you fully implemented file replication and O/S update, these same mechanisms can be used to perform client configuration management. If not, do something now. You must be able to maintain /etc/* without manually logging into machines, or you will soon be spending all of your time pushing out ad hoc changes.

Earlier, we mentioned the Carnegie Mellon Software Update Protocol (SUP). SUP replicated files for us. These files included the /etc/services file, automount maps, many other maps that are normally served by NIS, and the typical suite of gnu tools and other open-source utilities usually found in /usr/local on UNIX systems. In each case, we generalized what we could so every client had identical files. Where this was not practical (clients running cron jobs, clients acting as DNS secondaries, etc.), we applied a simple rule: send a configuration file and a script to massage it into place on the client's hard disk. SUP provided this "replicate then execute" mechanism for us so we had little need to add custom code.

In most cases we ran SUP from either a cron job or a daemon script started from `/etc/inittab`. This generally triggered replications every few minutes for frequently-changed files, or every hour for infrequently changed files.

The tool we used for managing client crontabs was something we wrote called ‘crontabber’ [crontabber]. It worked by looking in `/etc/crontab.master` (which was SUPed to all client machines) for crontab entries keyed by username and hostname. The script was executed on each client by SUP, and execution was triggered by an update of `crontab.master` itself. The `crontab.master` file looked something similar to Listing 2.

Step 13: Client Application Management

Prerequisites: Client Configuration Management.

Everything up to this point has been substrate for applications to run on – and we need to remember that applications are the only reason the infrastructure exists in the first place. This is where we make or break our infrastructure’s perception in the eyes of our users.

We wanted location transparency for every application running on any host in our “virtual machine.” We wanted the apparent location and directory structure to be identical whether the application was installed on the local disk or on a remote file server. To accomplish this, we used SUP to maintain identical installations of selected applications on local disks, automounted application directories for NFS-served apps, and Perl-managed symbolic link farms to glue it all together [mott].

A heterogeneous and readily available caching filesystem would have been much simpler to understand, and as mentioned before we originally considered AFS.

We made all applications available for execution on all hosts, regardless of where the application binaries physically resided. At first, it may seem strange that a secretary might have the ability to run a CAD

program, but an ASIC engineer will certainly appreciate the fact that, when their own workstation fails, the secretary’s machine can do the job (see the *Disaster Recovery* section).

We executed our apps from `/apps/application_name`. We had the automounter deliver these binaries, not to `/apps`, but to `/remote/apps/application_name`. We then created a symbolic link farm in `/apps`. The link farm simply pointed to the `/remote/apps` directories of the same name.

To support the extra speed we needed for some applications, we used SUP to replicate the application from the NFS server into the `/local/apps/application_name` directory on the client hard disk. The Perl code which drove SUP referred to a flat file (`/etc/autosup.map`) which listed applications to be replicated on particular machines. We inspiringly dubbed this code ‘autosup’ [autosup]. The `autosup.map` file looked something like:

```
scotty: elm wingz escapade metrics
luna:  elm wingz
[...]
```

After ‘autosup’ updated the local copies of applications, possibly adding or deleting entire apps, another Perl script, ‘autolink’, updated the symbolic link farm to select the “best” destination for each `/apps` symlink. The selection of the best destination was made by simply ordering the autolink targets (in `/etc/autolink.map`) so that more preferential locations overrode less preferential locations. The `autolink.map` file usually looked something like Listing 3. The trivial example in Listing 4 shows how the symbolic links in `/apps` would look with a CAD package installed locally, and TeX installed on a file server.

The ‘autosup’ script was usually triggered by a nightly crontab which SUPed down the new `autosup.map`, and ‘autolink’ was usually triggered by ‘autosup’.

It is important to note that part of application management is developer management. At first, many

```
# create      from
#
/apps         /remote/apps
/apps         /local/apps
/apps/pub     /remote/apps/pub
#
/prd/sw       /net/${HOMESERVER}/export/apps${HOE}/prd/sw
/prd/sw       /local/apps1/prd/sw
[...]
```

Listing 3: autolink.map file.

```
/apps/CAD----> /local/apps/CAD      /remote/apps/CAD (ignored)
/apps/TeX      ---->               /remote/apps/TeX
```

Listing 4: /apps link farm examples.

of our application developers loved to have their programs write files in the directory tree that contained their program, and they tended to hardcode pathnames to other binaries. We consider this a bad thing. For our in-house developers we managed to convince them to refer to environment variables for where data and binaries lived. For external applications we had to do tricks with symlinks.

Step 14: Mail

Prerequisites: Client Configuration Management.

Now that you have a way of managing sendmail.cf on client hard disks you can set up mail. Avoid like the plague any attempts to use non-SMTP mail solutions – the world has gone SMTP, and there are now many fine GUI SMTP mail readers available. Proprietary solutions are no longer necessary for user-friendliness. We used NFS-mounted mail spools: POP or IMAP would probably be the better choice today.

Step 15: Printing

Prerequisites: Client Configuration Management.

During the first few days after any new infrastructure went live, we usually spent about 80% of our time fixing unforeseen printing problems. Printers will eat your lunch. Assuming you can pick your printers, use high-quality postscript printers exclusively.

The best print infrastructure we've seen by far is the one a major router vendor uses internally – 90 Linux print servers worldwide spooling to 2000 printers, seamlessly and reliably providing print service to thousands of UNIX and NT clients via Samba [samba]. The details of this infrastructure have not been released to the public as of the time this paper goes to press – check www.infrastructures.org for an update.

Step 16: Monitoring

Prerequisites: Client Application Management.

When all of the above was done, we found very little monitoring was needed – the machines pretty much took care of themselves. We never got around to setting up a central syslogd server, but we should have. We only had paging working spottily at best. These days, with most alpha paging vendors providing free e-mail gateways, this should be much easier. Otherwise, you may want to take a look at the Network Paging Protocol (SNPP) support in HylaFAX. [hylafox]

Migrating From an Existing Infrastructure

Think of a migration as booting a new virtual machine, and migrating your old hardware into the new virtual machine.

The first infrastructure we used to develop this model was in fact one that had started chaotically, as four desktop machines that were administered by the application developers who sat in front of them. As

the internal application they developed became successful, the infrastructure grew rapidly, and soon consisted of 300 machines scattered worldwide. At the time we embarked on this effort, these 300 machines were each unique, standalone hosts – not even DNS or NIS were turned on. This state of affairs is probably all too typical in both large and small organizations.

If you are migrating existing machines from an old infrastructure (or no infrastructure) into a new infrastructure, you will want to set up the infrastructure-wide services (like NIS, DNS, and NFS) first. Then, for each desktop host:

1. Create a replacement host using your chosen "Host Install" tool as described in this paper.
2. Have the user log off.
3. Migrate their data from their old workstation to an NFS server.
4. Add the new NFS-served directory to the automounter maps so the new host can find it.
5. Drop the new client on the user's desk.

This may sound impossible if each of your desktop hosts have unique filesystem layouts, or still have a need to retain unique data on their own hard disk. But we were able to accommodate some of these variations with some thought, and get rid of the rest. Some of the ways we did this are described in the sections above.

We found it to be much easier and more effective in the long run to roll through an existing infrastructure replacing and rebuilding hosts, rather than trying to converge a few files at a time on the existing hosts. We tried both. Where we replaced hosts, a 100-host infrastructure could be fully converted to the new world order in under three months, with one sysadmin working at it half-time. User impact was limited to the time it took to swap a host. Where we instead tried to bring order out of chaos by changing one file at a time on all hosts in an infrastructure, we were still converging a year later. User impact in this case was in the form of ongoing and frustrating changes to their world, and prolonged waits for promised functionality.

Disaster Recovery

The fewer unique bytes you have on any host's hard drive, the better – always think about how you would be able to quickly (and with the least skilled person in your group) recreate that hard drive if it were to fail.

The test we used when designing infrastructures was "Can I grab a random machine and throw it out the tenth-floor window without adversely impacting users for more than 10 minutes?" If the answer to this was "yes," then we knew we were doing things right.

Likewise, if the entire infrastructure, our "virtual machine," were to fail, due to power outage or terrorist bomb (this was New York, right?), then we should expect replacement of the whole infrastructure to be

no more time-consuming than replacement of a conventionally-managed UNIX host.

We originally started with two independent infrastructures – developers, who we used as beta testers for infrastructure code; and traders, who were in a separate production floor infrastructure, in another building, on a different power grid and PBX switch. This gave us the unexpected side benefit of having two nearly duplicate infrastructures – we were able to very successfully use the development infrastructure as the disaster-recovery site for the trading floor.

In tests we were able to recover the entire production floor – including servers – in under two hours. We did this by co-opting our development infrastructure. This gave us full recovery of applications, business data, and even the contents of traders' home directories and their desktop color settings. This was done with no hardware shared between the two infrastructures, and with no "standby" hardware collecting dust, other than the disk space needed to periodically replicate the production data and applications into a protected space on the development servers. We don't have space here to detail how the failover was done, but you can deduce much of it by thinking of the two infrastructures as two single machines – how would you allow one to take on the duties and identity of the other in a crisis? With an entire infrastructure managed as one virtual machine you can have this kind of flexibility. Change the name and reboot...

If you recall, in our model the DNS domain name was the name of the "virtual machine." You may also recall that we normally used meaningful CNAMEs for server hosts – gold.mydom.com, sup.mydom.com, and so on. Both of these facts were integral to the failover scenario mentioned in the previous paragraph, and should give you more clues as to how we did it.

Push vs. Pull

We swear by a pull methodology for maintaining infrastructures, using a tool like SUP, CVSup, or 'cfengine'. Rather than push changes out to clients, each individual client machine needs to be responsible for polling the gold server at boot, and periodically afterwards, to maintain its own rev level.

Before adopting this viewpoint, we developed extensive push-based scripts based on rsh, rcp, and rdist.

The problem we found with the r-commands was this: When you run an r-command based script to push a change out to your target machines, odds are that if you have more than 30 target hosts one of them will be down at any given time. Maintaining the list of commissioned machines becomes a nightmare.

In the course of writing code to correct for this, you will end up with elaborate wrapper code to deal with: timeouts from dead hosts; logging and retrying

dead hosts; forking and running parallel jobs to try to hit many hosts in a reasonable amount of time; and finally detecting and preventing the case of using up all available TCP sockets on the source machine with all of the outbound rsh sessions.

Then you still have the problem of getting whatever you just did into the install images for all new hosts to be installed in the future, as well as repeating it for any hosts that die and have to be rebuilt tomorrow.

After the trouble we went through to implement r-command based replication, we found it's just not worth it. We don't plan on managing an infrastructure with r-commands again, or with any other push mechanism for that matter. They don't scale as well as pull-based methods.

Cost of Ownership

Cost of ownership is priced not only in dollars but in lives. A career in Systems Administration is all too often a life of late nights, poor health, long weekends, and broken homes.

We as an industry need to raise the bar for acceptable cost of administration of large numbers of machines. Most of the cost of systems administration is labor [gartner]. We were able to reduce this cost enough that, while the number of machines we were administering grew exponentially, our group only grew linearly. And we all got to spend more nights and weekends at home.

While we were unable to isolate any hard numbers, to us it appears that, by using the techniques described in this paper, systems administration costs can be reduced by as much as an order of magnitude, while at the same time providing higher levels of service to users and reducing the load on the systems administrators themselves.

SysAdmin or Infrastructure Architect?

There's a career slant to all of this.

Infrastructure architects typically develop themselves via a systems administration career track. That creates a dilemma. A systems administration background is crucial for the development of a good infrastructure architect, but we have found that the skillset, project time horizon, and coding habits needed by an infrastructure architect are often orthogonal to those of a systems administrator – an architect is not the same animal as a senior sysadmin.

We have found, in the roles of both manager and contractor, that this causes no end of confusion and expense when it comes to recruiting, interviewing, hiring, writing and reading resumes, and trying to market yourself. Recruiters generally don't even know what an "infrastructure architect" is, and far too often assume that "senior sysadmin" means you know how to flip tapes faster. Most of us at one time or another

have been restricted from improving a broken infrastructure, simply because it didn't fit within our job description.

In order to improve this situation, we might suggest that "infrastructure architect" be added to the SAGE job descriptions, and USENIX and affiliate organizations help promulgate this "new" career path. We'd like to see more discussion of this though. Is an IA an advanced version of a sysadmin, or are they divergent?

There seems to us to be a mindset – more than skillset – difference between a sysadmin and an architect.

Some of the most capable systems administrators we've known are not interested in coding (though they may be skilled at it). When given a choice they will still spend most of their time manually changing things by logging into machines, and don't mind repetitive work. They tend to prefer this direct approach. They can be indispensable in terms of maintaining existing systems.

As mentioned before, infrastructure architects tend to spend most of their time writing code. They are motivated by challenges and impatience – they hate doing the same thing twice. When allowed to form a vision of a better future and run with it they, too, can be indispensable. They provide directed progress in infrastructures which would otherwise grow chaotically.

While most people fall somewhere between these two extremes, this difference in interests is there – it may not be fair or correct to assume that one is a more advanced version of the other. Resolving this question will be key to improving the state of the art of enterprise infrastructures.

Conclusion

There are many other ways this work could have been done, and many inconsistencies in the way we did things. One fact that astute readers will spot, for instance, is the way we used both file replication and a makefile to enact changes on client disks. While this rarely caused problems in practice, the most appropriate use of these two functions could stand to be more clearly defined. We welcome any and all feedback.

This is the paper we wish we could have read many years ago. We hope that by passing along this information we've aided someone, somewhere, years in the future. If you are interested in providing feedback on this paper and helping improve the state of the art, we'd like to welcome you to our web site: Updates to this paper as well as code and contributions from others will be available at www.infrastructures.org.

Acknowledgments

Karen Collins has probably read this paper more times than any human alive. The good wording is hers

– the rest is ours. Her tireless attention to details and grammar were crucial, and the errors which remain are due to our procrastination rather than her proofreading ability. We'd hire her as a technical writer any time. Next time we'll start a month earlier, Karen.

Many thanks go to Pamela Huddleston for her support and patience, and for providing the time of her husband.

We were extremely fortunate in getting Eric Anderson as our LISA "shepherd" – many substantial refinements are his. Rob Kolstad was extremely patient with our follies. Celeste Stokely encouraged one of us to go into systems administration, once upon a time.

We'd like to thank NASA Ames Research Center and Sterling Software for the supportive environment and funding they provided for finishing this work – it would have been impossible otherwise.

Most of all, we'd like to thank George Sherman for his vision, skill, compassion, and tenacity over the years. He championed the work that this paper discusses. We're still working at it, George.

This paper discusses work performed and influenced by Spencer Westwood, Matthew Buller, Richard Hudson, Jon Sober, J. P. Altier, William Meenagh, George Ott, Arash Jahangir, Robert Burton, Jerzy Baranowski, Joseph Gaddy, Rush Taggart, David Ardley, Jason Boud, Lloyd Salmon, George Villanueva, Glenn Augenstein, Gary Merinstein, Guillermo Gomez, Chris York, Robert Ryan, Julie Collinge, Antonio DiCaro, Victoria Sadoff, James McMichael, Mark Blackmore, Matt Martin, Nils Eliassen, Richard Benzell, Matt Forsdyke, and many, many others over the course of four years, in three cities, spanning two continents. Best wishes to you all. If we missed your name we owe you dinner at Maggie's someday.

Author Information

Steve Traugott taught himself BASIC while standing up in Radio Shack in front of a TRS-80 Model I. At Paradyne he was a modem internals technician when modems were more expensive than cars, and dabbled in COBOL on an IBM/370 clone he built behind his desk. He decided to stop all of that when Challenger exploded within view of his office, and became an F-15 and AC-130 gunship crew chief in the U.S. Air Force to gain a better understanding of the aerospace industry. After realizing that aerospace needs better computing, he returned to civilian life at IBM to port OSF/1 to their mainframe family, worked on the last releases of System V UNIX at AT&T in 1993, and experienced DCE at Digital Equipment Corporation. He became a senior architect, then Vice President of trading floor infrastructure engineering for Chemical and Chase Manhattan banks, then escaped from New York for a contract at Cisco Systems. He has now found a home in the supercomput-

ing branch of NASA Ames Research Center, in Silicon Valley.

Joel Huddleston also taught himself BASIC while standing up in Radio Shack in front of a TRS-80 Model I. He began his computer career at Geophysical Services, Inc. operating a TIMAP II computer that still used punched cards for job entry. After a distinguished and lengthy career as a college student and part-time programmer/systems administrator at Texas A&M University, Mr. Huddleston entered the "Real World" when he discovered that students cannot be tenured. As a computer consultant for SprintParanet, Mr. Huddleston worked for such diverse firms as CompUSA, Motel 6, and Chase Manhattan Bank on projects ranging from a 100 seat Banyan/Netware migration to the design of 800+ seat Solaris trading floors.

References

- [afs] John H. Howard, Carnegie Mellon University, "On Overview of the Andrew File System," *USENIX Conference Proceedings*, Winter, 1988.
- [amanda] *Advanced Maryland Automatic Network Disk Archiver*, <http://www.cs.umd.edu/projects/amanda/index.html>.
- [anderson] Paul Anderson, "Towards a High-Level Machine Configuration System," *USENIX Proceedings: Eighth Systems Administration Conference (LISA '94)*.
- [athena] G. Winfield Treese, "Berkeley UNIX on 1000 Workstations: Athena Changes to 4.3BSD," *USENIX Conference Proceedings*, Winter, 1988.
- [autosup] *Automatic Application Replication Tool*, available from <http://www.infrastructures.org>.
- [bhide] A. Bhide, E. N. Elnozahy, and S. P. Morgan. "A Highly Available Network File Server," *Proceedings of the 1991 USENIX Winter Conference*, 1991.
- [bsd] *BSDlite 4.4, FreeBSD, NetBSD, OpenBSD Distribution Repository*, <http://www.freebsd.org/cgi/cvsweb.cgi>.
- [burgess] Mark Burgess, *GNU Cfengine*, <http://www.iu.hioslo.no/~mark/cfengine/>.
- [coda] *Coda Distributed Filesystem*, <http://www.coda.cs.cmu.edu>.
- [crontabber] *Client Crontab Management Script*, available from <http://www.infrastructures.org>.
- [cvs] *Concurrent Versions System*, <http://www.cyclic.com/cvs/info.html>.
- [dce] *OSF Distributed Computing Environment*, <http://www.opengroup.org/dce/>.
- [depot] Ken Manheimer, Barry Warsaw, Steve Clark, Walter Rowe, "The Depot: A Framework for Sharing Software Installation Across Organizational and UNIX Platform Boundaries," *USENIX LISA IV Conference Proceedings*, October 24-25, 1991.
- [dns] Paul Albitz & Cricket Liu, *DNS and BIND, 2nd Edition*, O'Reilly & Associates, 1996.
- [evard] Rémy Evard, "An Analysis of UNIX System Configuration," *USENIX Proceedings: Eleventh Systems Administration Conference (LISA '97)*, October 26-31, 1997.
- [hylafax] *Using HylaFAX as an SNPP Server*, <http://www.vix.com/hylafax/ixotap.html>.
- [libes] Don Libes, *Exploring Expect*, ISBN 1-56592-090-2, O'Reilly & Associates, 1994.
- [frisch] Aileen Frisch, *Essential System Administration, 2nd Edition*, O'Reilly & Associates, 1995.
- [gartner] Cappuccio, D., Keyworth, B., and Kirwin, W., *Total Cost of Ownership: The Impact of System Management Tools*, Gartner Group, 1996.
- [hagemark] Bent Hagemark, Kenneth Zadeck, "Site: A Language and System for Configuring Many Computers as One Computing Site," *USENIX Proceedings: Large Installation Systems Administration III Workshop Proceedings*, September 7-8, 1989.
- [limoncelli] Tom Limoncelli, "Turning the Corner: Upgrading Yourself from 'System Clerk' to 'System Advocate'," *USENIX Proceedings: Eleventh Systems Administration Conference (LISA '97)*, October 26-31, 1997.
- [lirov] Yuval Lirov, *Mission-Critical Systems Management*, Prentice Hall, 1997.
- [mott] Arch Mott, "Link Globally, Act Locally: A Centrally Maintained Database of Symlinks," *USENIX LISA V Conference Proceedings*, September 30-October 3, 1991.
- [nemeth] Evi Nemeth, Garth Snyder, Scott Seebass, Trent R. Hein, *UNIX System Administration Handbook, second edition*, Prentice Hall, 1995.
- [ntp] *Network Time Protocol Home Page*, <http://www.eecis.udel.edu/~ntp/>.
- [polstra] John D. Polstra, *CVSup Home Page and FAQ*, <http://www.polstra.com/projects/freeware/CVSup/>.
- [rabbit] 'rabbit' *expect script for ad hoc changes*, available from <http://www.infrastructures.org>.
- [rpm] *Red Hat Package Manager - open source package tool*, <http://www.rpm.org>.
- [rudorfer] Gottfried Rudorfer, "Managing PC Operating Systems with a Revision Control System," *USENIX Proceedings: Eleventh Systems Administration Conference (LISA '97)*, October 26-31, 1997.
- [samba] *CVS Access to Samba Source Tree*, <http://samba.anu.edu.au/cvs.html>.
- [shafer] Steven Shafer and Mary Thompson, *The SUP Software Upgrade Protocol*, <http://www.cs.cmu.edu/afs/cs.cmu.edu/project/mach/public/sup/sup.ps>, 8 September, 1989.
- [stern] Hal Stern, *Managing NFS and NIS*, O'Reilly & Associates, 1991.

[stokely] Extensive UNIX resources at Stokely Consulting – thanks Celeste!, <http://www.stokely.com/>.

[sup] *Carnegie Mellon Software Upgrade Protocol*, <http://www.cs.cmu.edu/afs/cs.cmu.edu/project/mach/public/sup/sup.tar.gz>, <ftp://sunsite.unc.edu/pub/Linux/system/network/management/sup.tar.gz>.

Ganymede: An Extensible and Customizable Directory Management Framework

Jonathan Abbey and Michael Mulvaney – The University of Texas at Austin

ABSTRACT

In the fall of 1994, Applied Research Laboratories, The University of Texas at Austin (ARL:UT) presented a paper [1] at LISA VIII, describing work that we had performed designing and implementing a management framework for NIS and DNS, called GASH. In the years since that paper was presented, it has become clear that the design of GASH was insufficient to meet the complex, idiosyncratic, and rapidly changing needs of modern networking. GASH suffered from being too inflexible to be rapidly retooled for a changing network environment, from being limited to a single user at a time, and from being unable to provide management services to custom clients.

In the face of these issues, the Computer Science Division at ARL:UT went back to the drawing board and developed a Java-based directory management framework on the basis of the design principles presented in our GASH paper. Written in Java, Ganymede¹ is based on a distributed object design using the Java Remote Method Invocation [2] protocol and features a multi-threaded, multi-user server, and a graphical, explorer-style client. By supporting customization through a graphical schema editor, plug-in Java classes, and external build scripts, Ganymede is able to support a variety of directory services, including NIS, DNS, LDAP, and even NT user and group management.

Introduction

In early 1992, our laboratory had a problem. We had a need to pull computers from all over the lab together into a common NIS [3] and DNS [4] regime, but the lab was separated into several roughly autonomous groups. We needed a way to create a centralized NIS and DNS domain while preserving the ability of the groups to control their own user and group accounts. In addition, we needed to centralize email delivery and administer automounter volume definitions to support a useful and transparent network architecture. In response to all this, the Computer Science Division at ARL:UT developed the Group Administration Shell (GASH, for short), a text-based shell that allows designated users in our laboratory to issue commands that modify our NIS and DNS tables. We put this into operation in late 1993, and for the last four and a half years, we have run our laboratory on GASH, enjoying significant benefits in database consistency and simplicity of administration.

There have been problems with GASH, however. GASH was a very rigid program that directly manipulated NIS source files and a complex system database file that was transformed into DNS by a Perl script. Whenever we had a need to alter or elaborate any aspect of our network computing environment, we found that modifying GASH was extremely difficult and time-consuming. In addition, certain aspects of

GASH's operation proved troublesome in practice. The permissions and ownership model used by GASH was very idiosyncratic, and made it difficult to transfer users between groups. More fundamentally, GASH had no clean way to interact with other tools. If a user wanted to change his password, he had to have his GASH administrator change it with GASH, or use `yppasswd` and take the chance that the `yppasswd` daemon might conflict with an administrator making changes in GASH. GASH was clearly an inadequate tool to take us into the future. We wanted to be able to tie our network and account management tools into our personnel databases, we wanted to be able to modify our network topology as needed without spending six months reworking the 50,000 lines of C code in GASH each time, we wanted to support LDAP and NT, and we wanted all of our end users to be able to take advantage of our management tools, which wasn't practical with a single-user tool.

By late 1995, we knew something had to be done. Looking around, we were not able to find a suitable and reasonably priced commercial tool that was focused on the issues we had developed GASH to address and that would give us the path to the future we wanted. The network management packages we were aware of at the time were all focused on managing distributed workstations rather than managing centralized directory services.

We had a lot of experience and insights into the problem domain we were dealing with. We knew we wanted a client-server system. We knew we wanted a generic system that could be easily customized, and

¹Which stands for The "GASH Network Manager, Deluxe Edition," of course.

we knew we wanted a GUI. So, our task was set. We would work to build a GUI GASH Construction Set.

A GUI GASH Construction Set

In our LISA VIII paper [1], we observed that we saw some potential in reworking GASH around an object database [5] to facilitate automatic consistency maintenance and the provisioning of a GUI. By late 1995, a certain highly caffeinated object oriented programming language was making a lot of noise for its sophistication, ease of use, and portability. After doing an extensive design review, we determined that Java looked as though it would enable us to meet our design goals on all fronts. The Java Virtual Machine provided us with the ability to have our code run on PC's and Mac's as well as on our UNIX workstations and X terminals, and the Java Remote Method Invocation (RMI) protocol allowed us to do a true distributed object design [6] without having to worry about obtaining a CORBA ORB² for all the machines in our laboratory.

After spending the first half of 1996 doing design work with pen and paper, we began the work of implementing Ganymede. Two years and 140,000 lines of Java later, in mid-1998, we are currently beta-testing the Ganymede system. We have produced a robust and customizable client/server system capable of doing everything GASH did, with plenty of room to grow. Our beta-testers have run the Ganymede server on Sparc Solaris, AIX, Linux, and FreeBSD. The client has been run successfully on Windows 95 and NT using Sun's Java browser plug-in and from the command line on the UNIX platforms mentioned above. At the lab, we are running Ganymede on a test basis with all the data from our installation of GASH loaded into the server and experiencing full functionality and essentially perfect up time on the server under the 1.2 beta 4 JDK. We expect to have fully transitioned to running the lab on Ganymede by the time of the LISA 1998 conference.

What Is Ganymede?

Ganymede is a system for managing data which is to be fed into a network through some standard distribution mechanism, such as NIS, DNS, Rdist, LDAP, or the NT Domain Controller system. Ganymede is designed with an emphasis on providing tight control over what types of changes can be made to the database it manages, and on allowing multiple users to make changes to that database simultaneously. The

Ganymede server is not intended to serve as a high-volume directory server, but rather is designed to work with directory systems designed for high-volume use, such as NIS, DNS, and LDAP, whose servers may not themselves provide useful mechanisms for managing changes.

The Ganymede system is based on a client-server design. The server contains a built-in object database, and supports custom Java plug-ins which provide intelligent management of object types defined in the server. The server supports an admin console which can monitor the server and that includes a GUI schema editor that can alter the definition of the database held in the server as the server runs. Several clients can be talking to the Ganymede server simultaneously, each browsing the database, issuing queries, making changes, and committing transactions without interfering with each other. Whenever a client commits changes to the database, the server can schedule one or more custom builder tasks to write out source files for NIS, DNS, or whatever is being supported, and then run an external script to propagate the exported data into the network environment. See Figure 1 for a diagram of the overall system.

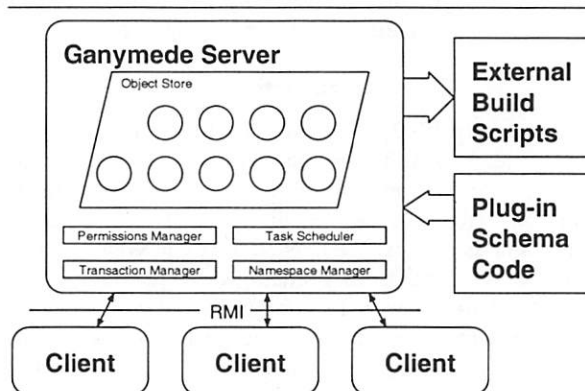


Figure 1: Ganymede Block Diagram.

The Ganymede Server

The Ganymede server, like the rest of the Ganymede system, is written entirely in Java. It contains over 100 classes, which provide for the storage and manipulation of objects, the management of object locking and transactions, the scheduling of database checks and external build processes, and a comprehensive ownership and permissions model, among other features.

Object Store

The Ganymede server has a built-in object database which is held in memory while the Ganymede server runs. The objects in the database are held in a set of thread-synchronized hashing data structures. This design gives the server good performance and multi-threaded safety at the cost of a potentially large RAM footprint. The in-memory database is backed by an on-disk **ganymede.db** file. During

²CORBA stands for the Common Object Request Broker Architecture. It is a standard for allowing object oriented code to communicate, object-to-object, across networks. An ORB is an Object Request Broker, a piece of software that handles the network communications on behalf of object oriented code on a given system. The CORBA specification is a product of the Object Management Group (OMG), and more information can be found at their website [7].

execution, the server dumps its database to `ganymede.db` every two hours. Between dumps, the server maintains a **journal** file, which is a record of transactions made since the last database dump.

The Ganymede database is broken down by object type. Everything held in the database is an instance of a defined object type. Such object types might include things like "User," "Group," "System," and so on. There are a number of object types pre-defined in the database that the server depends on for its operation. These are shown in **Table 1**, below. We will discuss all of these built-in object types in more detail later. For now, notice that each object type in the database is identified by an object type code. The server can handle Object Type ID's from 0 to 32k, but Type ID's of 256 or less are reserved for the server's internal use. Object Type ID's above 256 are available for Ganymede adopters to define for their own use.

Object	Object Type ID
Owner Group	0
Admin Persona	1
Role	2
User	3
System Event	4
Builder Task	5
Object Event	6

Table 1: Mandatory Object Types.

Each type of object in the Ganymede database has its own set of data fields defined. Each field has a name and ID, and can have certain options defined, depending on the type of field. The Ganymede server supports eight different types of fields, as shown in **Table 2**. Most types of fields just hold a single value, but String, IP Address, and Object Reference fields can be defined to be vectors, holding up to 32k values in a single field. String, Integer, and IP Address fields can be connected to a **namespace** defined in the server. The server manages such fields to make sure that the values in them are kept unique across the relevant objects and fields.

Field Type	Options
String	Vector, Length, Chars Allowed
Integer	Max/Min Value
Password	Crypted/Non-Crypted
Date	Max/Min Value
Boolean	Labeled/Non-Labeled
Permission Matrix	
IP address	Vector, IPv4 or IPv6
Object Reference	Vector, Target Type

Table 2: Field Types.

Most of these field types are self-explanatory, but a couple require some discussion. The **permission matrix** field type is used by the Ganymede

permissions system and is not really useful in any other context. We'll talk about where the server uses the permission matrix field type when we discuss the Ganymede permissions system. We do need to talk about the **object reference** field type, but before we get into the details of this field type we need to talk about how objects in the server are identified.

Invids and Invid Fields

Objects in the database are identified by an object called an **invid**, which stands for **IN**variant **ID**. An invid is a Ganymede object identifier, and is implemented as a pair of numbers. The first number is the object's type id, the second is a number between 0 and 2 billion unique to that object within its object type. Object numbers are never re-used. This makes it possible to unambiguously track the history of an object in the server's logs, but it does limit the server to handling 2 billion objects of a particular type over its lifetime.

The object reference field type is simply a field that holds invids. In fact, from now on, we'll refer to this field type as an **invid field**. Invid fields are used throughout Ganymede to link objects together. When one object's invid is placed in an invid field in another object, those objects are said to be linked. All object links in Ganymede are symmetrical, so that each object has references to all objects in the database that point at it, and vice versa. Because all objects in the database are symmetrically linked, the database can easily be kept up-to-date whenever objects are deleted. All that the server has to do in order to clean up after deleting an object is to modify all objects that were listed in the invid fields of the deleted object; it is not necessary to sweep through the entire database looking for linked objects. Another advantage of using invids to link objects is that objects in the database can be renamed or relabeled without disturbing the linkages established in the server.

Invid fields can be configured so that an invid field in one object is linked to an invid field in another object. This is shown in the **schema editor** screen shot in **Figure 2**. In this screen shot, we see the users field in the group object being edited. The users field in the group object is an invid field that points to the groups field in the user object. When the client edits a group object, the server will automatically provide a list of users that can be placed in this field. Adding a user to this group will automatically cause the group to be added to the user, and vice versa. The user can look at either object and see the relationship. This bi-directional linking is very important to the structure of the Ganymede server. It is responsible for a lot of the intelligence of the server. If a user were to try to delete a group, but didn't have permission to edit the users listed in the group, the server would detect this and might reject the operation, depending on how the schema was configured.

Some object reference fields are “edit-in-place,” which means that the objects referenced by that field are handled as if they were contained within the referencing object. An object type must be designated in the schema editor as an **embedded** object in order to be linked to an edit-in-place field. An embedded object is for the most part very much like any other object in the database, but it does not have its ownership and permissions tracked independently of its parent, and the server handles its creation and deletion a little bit differently. The easiest way to see the differences between embedded and top-level objects is by looking at the special fields the server uses to keep track of these objects.

Mandatory Fields

Just as there are mandatory object types, so too are there a number of mandatory field definitions. The fields shown in **Table 3** are defined in all non-embedded object types in the server. All field ID's below 100 are reserved for global fields (fields defined in all top-level objects). Field ID's between 100 and 256 are devoted to fields in the mandatory object types that the server depends on for its operations. Field ID's above 256 are user-assignable fields and can be configured in the schema editor.

Embedded objects have a different set of mandatory fields, which are shown in **Table 4**. The **container field** is simply a pointer to the object that the embedded object is contained in. This field is linked to the field in the parent where the embedded object appears.

Field	Type	Field ID
Owner List	Invid Vector	0
Expiration Date	Date	1
Removal Date	Date	2
Notes	String	3
Creation Date	Date	4
Creator Identifier	String	5
Last Modification Date	Date	6
Last Modifier Identifier	String	7
Back Links	Invid Vector	8

Table 3: Mandatory Fields For Top-Level Objects.

Field	Type	Field ID
Container	Invid	0
Back Links	Invid Vector	8

Table 4: Mandatory Fields For Embedded Objects.

Most of the mandatory fields should need no explanation. We'll discuss the **owner list** field when we talk about the Ganymede permissions system a bit later on. For now, let's talk about the **back links** field.

The back links field is a bit special. Previously, we said that the server guarantees that all references made in an invid field are symmetrical, and we gave the example of user and group object types having their groups and users fields symmetrically linked. This kind of visible bi-directional linking sometimes doesn't make sense. In cases where it doesn't, an invid field can be configured so that it points to an object without specifying the target field. The server will use

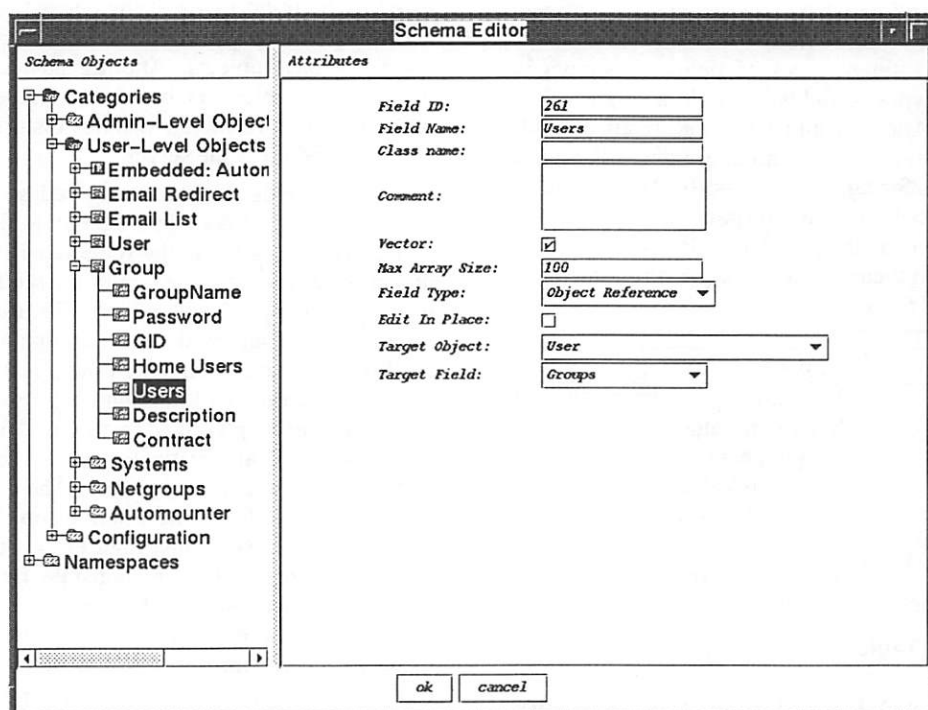


Figure 2: The Schema Editor Editing an Invid Field.

the back links field to handle the back-reference. The client won't see this back-link, but the server will, and will use it when the referenced object is deleted.

The DBEditObject Class and Server Customization

One of the keys to the Ganymede server's flexibility is that it takes advantage of Java's object oriented language features and dynamic linking to allow individual customizers to write classes to manage objects. The **DBEditObject** class in the server is consulted on every major decision having to do with how the server should handle operations on an object. The Ganymede schema editor allows adopters to bind

custom **DBEditObject** subclasses with object types in the server. **Figure 3** shows the **arlut.csd.ganymede.custom.userCustom** class being bound to the user object.

DBEditObject provides over two dozen methods that can be overridden by custom logic to inject intelligence into the Ganymede server. While it is out of the scope of this paper to describe all of the ways in which **DBEditObject** can be customized, we can mention a few highlights, and we will provide a simple sample of customization through **DBEditObject** subclassing.

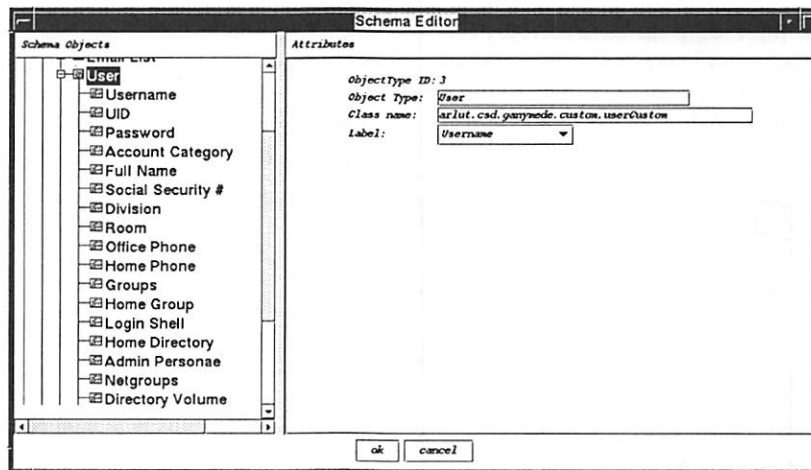


Figure 3: The Schema Editor Binding a Custom Class.

```
package arlut.csd.ganymede.custom;
import arlut.csd.ganymede.*;
public class userCustom extends DBEditObject {
// Boilerplate Constructors Omitted
public boolean fieldRequired(DBObject object,
                             short fieldid)
{
    switch (fieldid)
    {
        case userSchema.USERNAME:
        case userSchema.UID:
        case userSchema.HOMEDIR:
        case userSchema.LOGIN_SHELL:
            return true;
        case userSchema.PASSWORD:
            return !object.isInactivated();
    }
    return false;
}
}
```

Figure 4: A Simple Custom **DBEditObject** Subclass.

The DBEditObject class provides the ability for custom code to extend or override the default permissions system, to approve or deny any change to fields within an object based on the contents of the object, its relations with other objects, or the identity of the admin seeking to make the changes. It can provide a list of valid choices for **string** and **invid** fields. It can return custom dialogs in response to attempted operations, or even involve the client in a step-by-step wizard interaction sequence. It can get involved when a transaction is committed, to take actions outside of the database, such as creating home directories when users are created, or connecting changes to an object in the Ganymede database to an external database.

For a simple example of what is involved in a DBEditObject subclass, see **Figure 4**. This example shows the code necessary to specify what fields must be present in a user object when a transaction is committed.

Permissions and Ownership

One of the critical elements of Ganymede's design is the permissions model. Ganymede provides a universal permissions model that allows complete flexibility in apportioning privileges to classes of users/administrators, without becoming so unwieldy as to be impractical. The model is designed to support group administration, with ownership over objects shared by groups of administrators. Different classes of administrators can be defined, each with different privileges over different kinds of objects, and different fields within those objects. The Ganymede server has three object types defined to support this model, the **admin persona**, **owner group**, and **role** objects, as shown in **Figure 5**.

Each user in the Ganymede system can have one or more admin personae associated with it. The admin personae represent administrative privilege sets that the user can select, through an **su-like** mechanism in

Edit: Admin Persona - broccol:GASH Admin

File

General Owner Notes History Admin History

Name: broccol:GASH Admin

Password: [] []

Owner Sets

Selected	Available
ONG	ADG
	ASG
	Convex Shadow...
	EVG
	General/Direc...
	ITG
	Pequod
	SGG

Add

Roles

Selected	Available
GASH Admin	Admin Group H...
Secretary	Captain

Add

Admin Console ☒

Full Console ☐

Email Address: []

Figure 6: Ganymede Admin Persona.

the client. An admin persona object marries a list of owner group objects and a list of role objects to define the objects and operations that can be performed. The owner group objects define what objects are considered to be under the ownership of that admin persona, while the role objects define what operations the admin persona is permitted to perform.

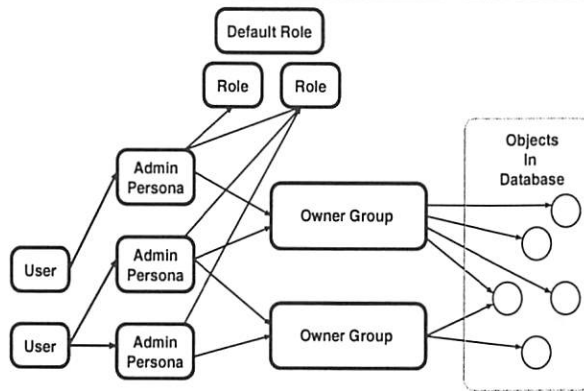


Figure 5: Ganymede Permissions Objects.

Admin Personae

All users registered with Ganymede may have some minimal permissions granted them by the **default role** object. This will typically include the ability to edit their own passwords, shells, and finger information, and to view some information about other users registered in the Ganymede database. If an individual is to have more privileges than that, an admin persona must be created for the user, as shown in Figure 6.

An admin persona includes a password, which must be entered by the user in order to access their extended privileges, a list of owner groups and roles, check boxes indicating whether the admin is privileged to run the server-monitoring console, and an optional email address for Ganymede to use to send mail to in response to the admin's actions.

Owner Groups

As mentioned above, all objects in the database are owned by owner groups, rather than by individual administrators. This design decision came out of our experience with GASH. By having all ownership vested in owner groups rather than in individual

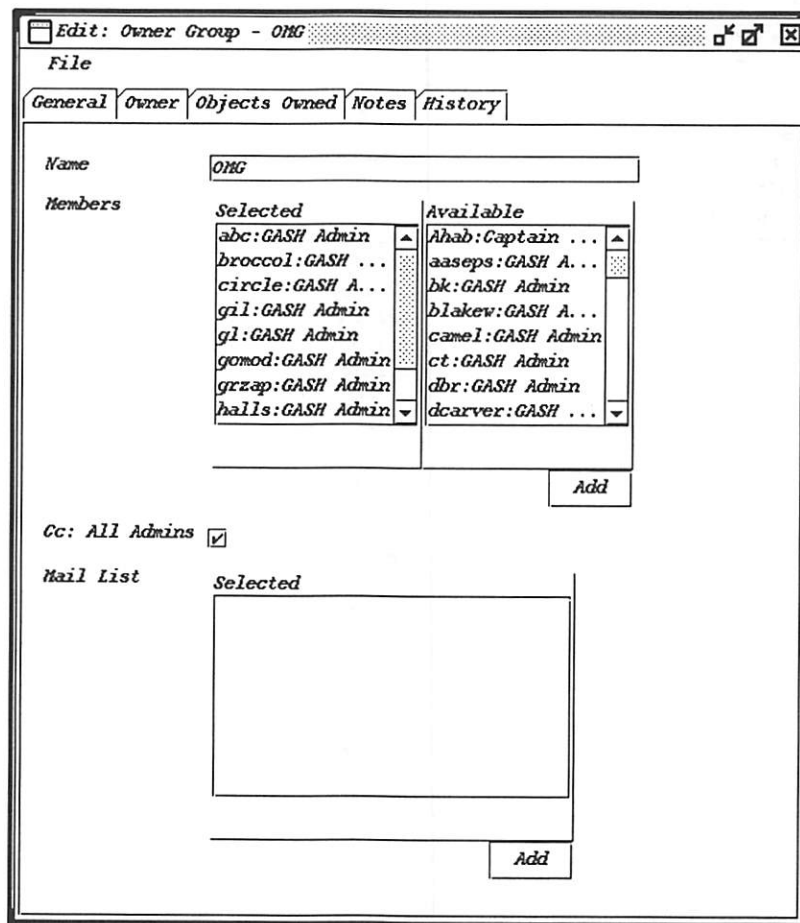


Figure 7: Ganymede Owner Group.

administrators, it is possible to bring a new administrator into a group without having to manually add that administrator to the ownership list of hundreds or thousands of objects. Other fields defined in the owner group object (see Figure 7) are designed to let administrators in a group share email notification for actions taken on objects owned by that owner group.

The Ganymede server supports a hierarchy of owner groups. One owner group can own another. Not only do the admins in the first owner group have ownership rights over the second, but also over all objects owned by that owner group. Figure 8 illustrates this. Admins belonging to the engineering owner group in Figure 8 own not only the hardware and software owner groups, but also all of the objects owned by those groups. In addition, owner groups are considered to own **themselves**, so an admin belonging to the engineering owner group could, if permitted by the roles granted to him, add or delete admins from the engineering group as well as the hardware and software groups.

The **supergash** owner group is special; all objects in the database, including all owner groups, are implicitly owned by the supergash owner group. Any admins belonging to the supergash owner group have “root” privileges over the Ganymede database.

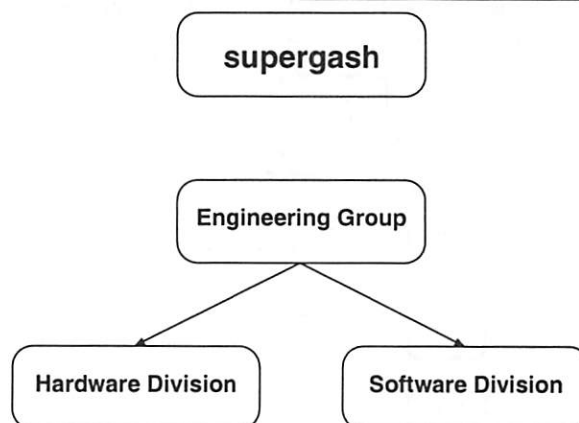


Figure 8: A Hierarchy of Ganymede Owner Groups.

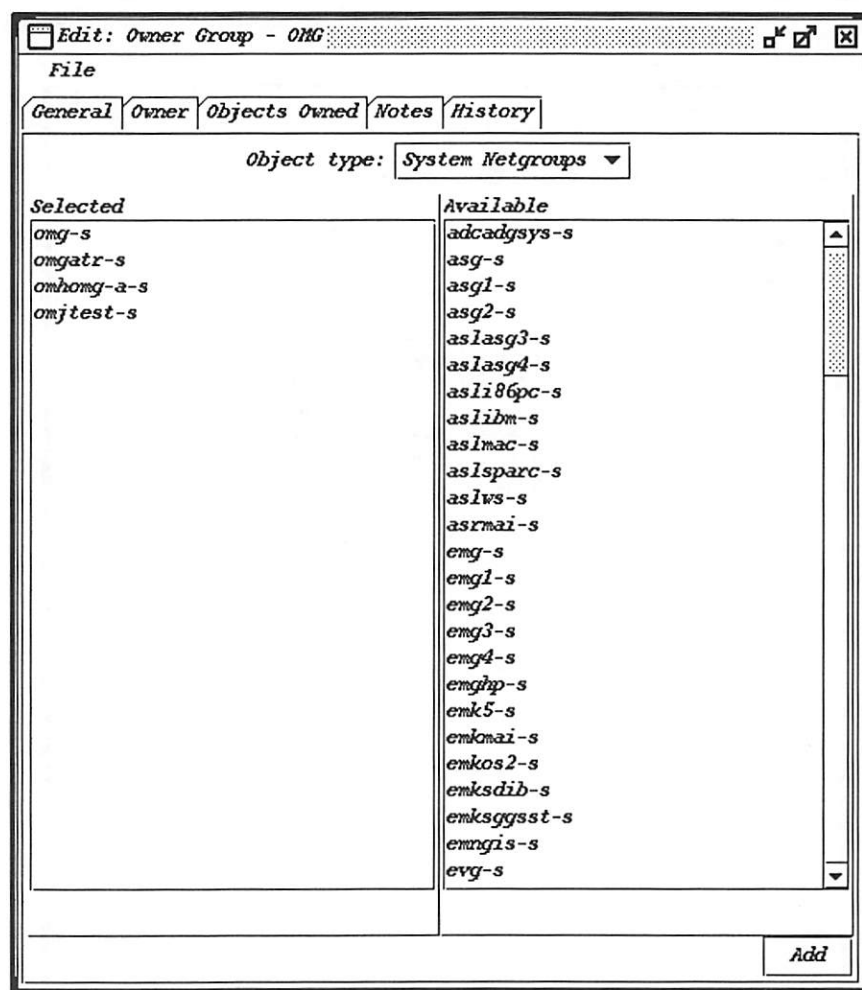


Figure 9: System Netgroup Objects Owned by an Owner Group.

An admin with editing privileges for an owner group can add or remove objects from that owner group by simply editing the owner group. This is shown in Figure 9.

Roles

The role object defines what an admin persona can do, both for objects owned by the admin through his owner group membership, and for objects "at large" in the database. Each role object contains two **permissions matrices**: one for objects owned, and one for default permissions. These permission matrices contain an array of booleans which allow access to the database by object type and field, with create, edit, view, and delete permissions categories. An admin can only view, edit, create or delete objects and fields that are specifically granted him by his set of roles. See Figure 10 for a list of fields defined in the role object.

Of special interest in Figure 10 is the "Delegatable Role?" check box. In order to support a true hierarchy of administrative control, admins can be granted the power to create new roles, and to create new admins. An admin who creates a new role or admin may not grant that role or admin privileges that he himself does not have from a delegatable role. That

is, if an admin has the **GASH admin** and **secretary** roles, and only the **secretary** role is delegatable, the admin will only be able to grant the **secretary** role to admins that he creates, and if he creates a new Role, will only be able to set bits in that Role that he got from either the **secretary** or **default** roles. Figure 11 demonstrates this. The check boxes that are visible correspond to bits that the admin editing the role has himself had granted to him through a delegatable role. The boxes X'ed out represent privileges that this admin may not grant to other roles.

The combination of the owner groups, which determine which objects are accessible, and the roles, which determine what can be done to those objects, provides complete flexibility while maintaining the ability to make wide-ranging changes in the authorization schema by simply editing one or two objects in the Ganymede database.

As implied above, all of the objects in the Ganymede server, including the owner group, role, and admin persona objects, are administered by this permissions system. The same permissions system that controls access to the Ganymede database also controls access to the controls themselves.

Edit: Role - GASH Admin

File

General Owner Notes History

Name:

Delegatable Role? ☐

Owned Object Bits:

Default Bits:

Persona entities

Selected	Available
aaseps:GASH A...	Ahab:Captain of...
abc:GASH Admin	monitor
bk:GASH Admin	mulvaney:Group ...
blakew:GASH A...	supergash
broccol:GASH ...	
camel:GASH Admin	
circle:GASH A...	
ct:GASH Admin	

Figure 10: A Ganymede Role.

As powerful as this system is, it is not complete. There will be cases where a more specialized permissions model is required. Take for instance the case of maintaining a public mailing list where users should be able to add and remove themselves, but not touch any other user in the list. All that would be required to support this sort of model would be to bind a custom DBEditObject subclass to the Mail List object type in the Ganymede server and redefine the **anonymousLinkOK()** and **anonymousUnlinkOK()** methods in DBEditObject. Other methods in the DBEditObject class can be overridden to implement custom ownership determination logic, or even to entirely override the normal persona/role/owner group permissions system.

Transactions

The Ganymede server is built around a transactional model wherein clients connected to the server check out objects for editing. There may be many clients connected to the server simultaneously, but changes made to objects in one transaction will not be visible to other users until the transaction is committed. Queries issued by clients are guaranteed to be atomic with respect to transactions across the duration of their processing.

The server supports checkpointing and rollback within the course of a transaction. This allows for complex sequences of operations to be attempted and undone if the sequence could not be carried to completion successfully.

The transaction commit process in the Ganymede server is based on two-phase commit logic, so that custom code can be written to connect

transactions issued in Ganymede to transactions in external databases.

In the Ganymede server, whenever transactions are committed, a record of the transaction is written to a journal file. This journal file allows the server to recover any transactions that were committed between the time that the server last performed a full database dump and an abnormal shutdown. The worst case for the Ganymede server on power failure or server crash is the loss of the most recently issued transaction.

When transactions are committed, the Ganymede scheduler schedules external build processes for execution. If multiple transactions are committed while the Ganymede scheduler is still executing the previous external build, the Ganymede scheduler will simply initiate another external build when the first build completes. Thus, multiple transactions made by users may be propagated to the external environment in bulk, depending on the rate that transactions are committed and the time necessary to complete an external build.

Logging and Email Notification

Ganymede, like GASH before it, has a very thorough logging and email notification system. All significant events on the server are logged to disk, and can also be emailed to interested parties. There are a wide variety of **system events** built into the server, and a supergash-level administrator can customize the server's email notification behavior. **Figure 12** shows the notification options for a system event.

While the system event categories are more-or-less hard-coded into the server, it is important that custom object types can have their significant events

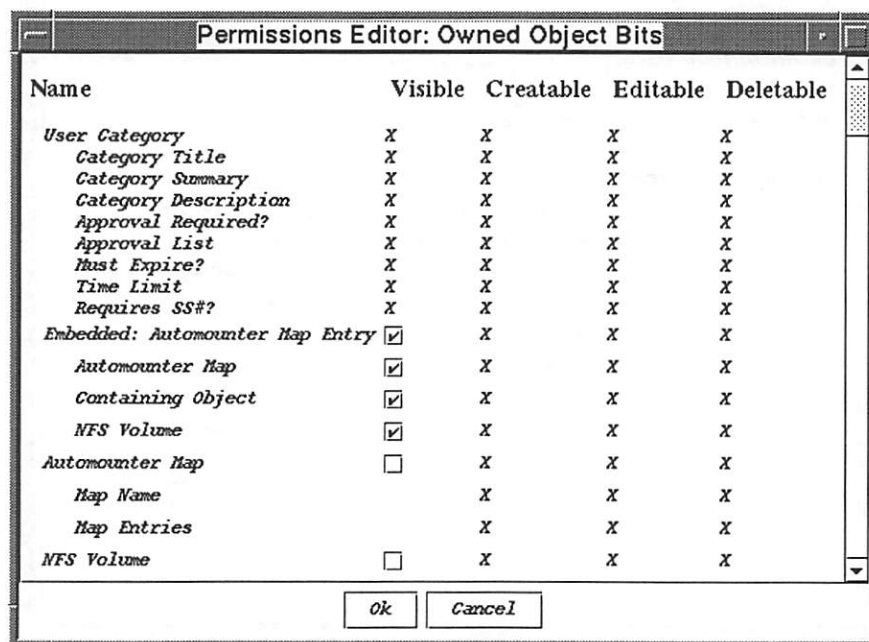


Figure 11: Permission Bits in a Role.

handled specially as needed. To support this, the Ganymede server also supports **object events**, which meld one of a set of common happenings to objects along with the type of object of interest. **Figure 13** tells the story.

One of the lessons we learned from GASH is that it is often important for admins to be able to tell what has happened to objects under their control. In response to this, the Ganymede log file has been designed to be easily parsed. The server itself can scan its own log file to report on changes made to an object, as shown in **Figure 14**.

Background Tasks

The Ganymede server includes a built-in task scheduling facility, similar to **cron**. A background thread queues various tasks for execution, including those shown in **Figure 15**.

The “garbage collection” task runs every night at midnight to do a bit of preemptive house cleaning in the server. The “database dumper” task runs every two hours, consolidating the database and cleaning the journal file. The “expiration” and “warning tasks” each run once a day, to handle objects that have had

their expiration or removal times set. The warning task is responsible for looking at objects that will expire or be removed in the near future, and sending out warnings to the administrators responsible for those objects through email. And then, there are the builder tasks.

Builder Tasks

Just as it is possible to define custom subclasses of **DBEditObject** to provide custom management of object types in the server, it is also possible to define custom builder tasks to be loaded into the server at runtime, as in **Figure 16**. Whenever a transaction is committed, any builder tasks registered with the server are scheduled for execution. The builder tasks will scan the Ganymede database, write out source files for NIS, DNS, etc., and call an external shell script to take those source files and propagate the data into NIS, DNS, or whatever else is being managed.

Schema Kits

The combination of a Ganymede schema definition, a set of custom **DBEditObject** subclasses, and any custom builder tasks together comprise a Ganymede **schema kit**. As currently available for

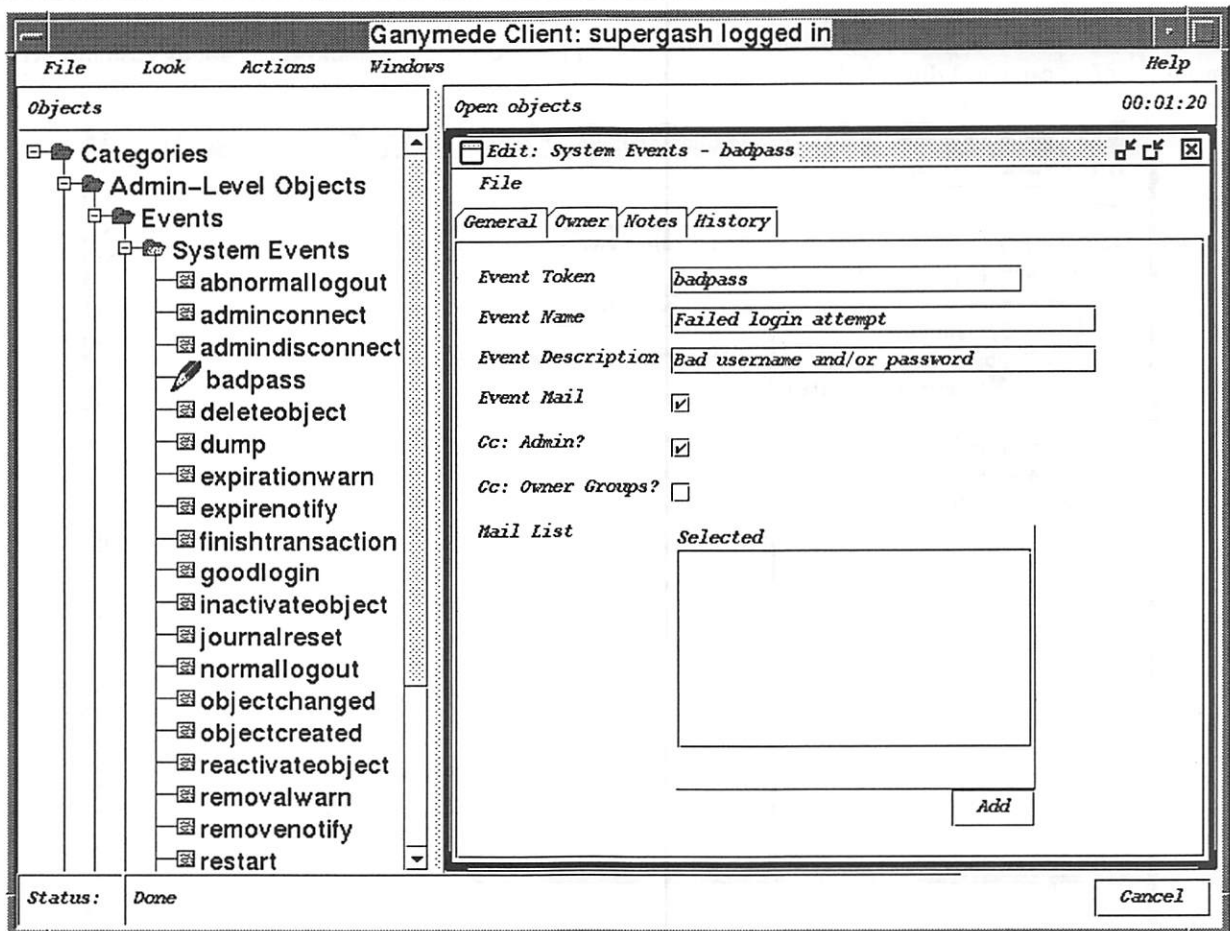


Figure 12: A System Event Record.

download, the Ganymede server includes three schema kits. One is an "nisonly" kit that handles Solaris-style passwd and group files, another is a "bsd" kit for managing BSD 4.4 master.passwd and group files, and the third is a full-scale "gash" kit that was designed as a way for adopters of GASH to have a minimal-work "drop-in" replacement for their existing GASH installations. Each of these kits includes a loader program to scan the original files and create a ganymede.db file which Ganymede can then manage.

The nisonly and bsd kits are provided so that UNIX admins can download Ganymede, load their passwd and group files, and start playing with Ganymede without a large commitment of time. The GASH kit is a much richer environment, and implements the complex network management logic described in our LISA VIII paper [1]. The GASH kit, like GASH itself, is designed to manage a single DNS domain and a single NIS domain, with support for NIS-specific features like netgroups and automounter configuration maps.

We are working on developing a next-generation schema based on the GASH model. This schema is to have richer support for DNS, including support for generic subnetwork allocation. We also want to separate out personal identification from the user account. Having separate person objects would facilitate proper generation of a canonical LDAP directory for our

laboratory, and would allow us to track responsibility for user accounts more conveniently.

Another design feature of our next-generation schema is support for NT and UNIX integration. We are currently shadowing our UNIX accounts into our NT domain controller using Rsh and some Perl scripts on the NT side. In our next-generation schema we plan to have a check-box on user objects to control whether or not the account should be replicated on NT, and to support NT group accounts as well as UNIX group accounts within Ganymede.

There are quite a number of other interesting possibilities for schema development. One obvious possibility is a DNS-only schema, with support for managing a large number of DNS domains and IP address ranges. We have demonstrated in our work implementing the GASH kit in Ganymede that the DBEditObject customization hooks are adequate to handle multiple-interface systems and network allocation. Creating a DNS-only schema kit would be some work, but we do not believe that the task would be overly difficult, and such a schema kit could be of considerable utility for ISP companies.

The Ganymede Client

The Ganymede server supports a generic Java interface for clients, allowing various custom clients and automated processes to talk to the server. After

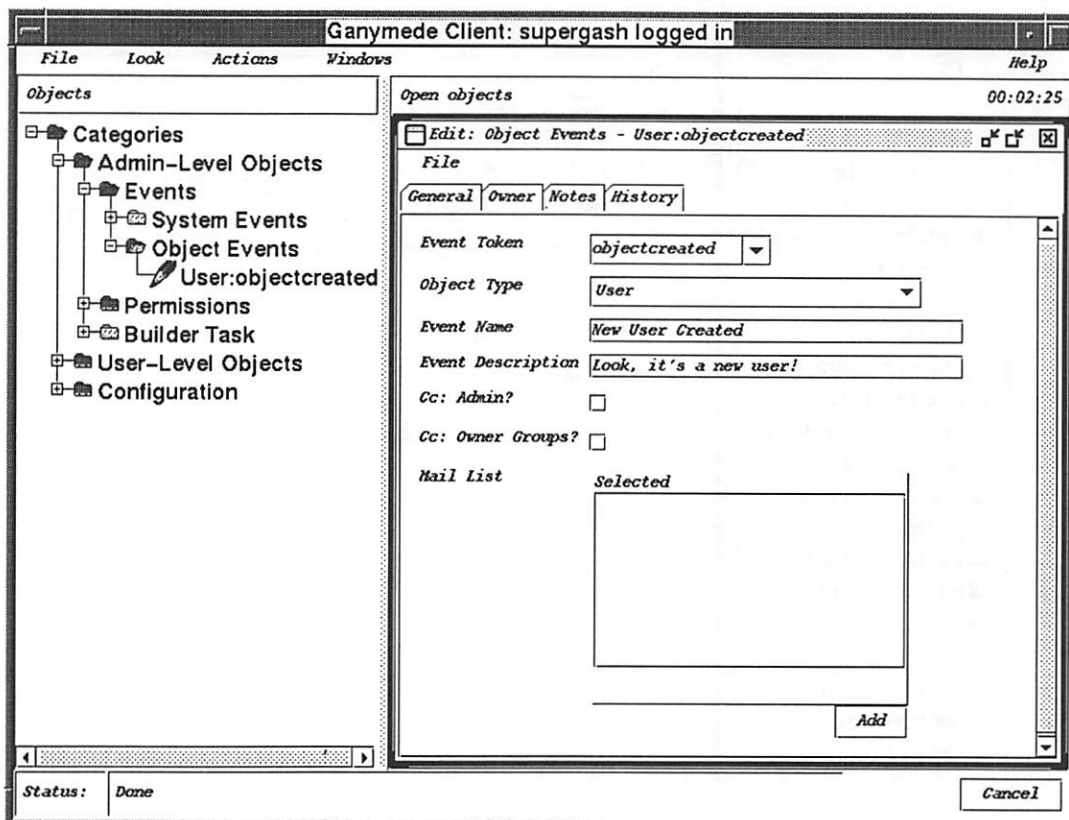


Figure 13: An Object Event Record.

connecting, clients can manipulate objects through RMI references to the objects in the database.

The generic client API permits us to write a variety of custom clients to handle specific tasks. For example, we have developed a command-line-based application that changes a user's password and looks just like Unix's `passwd` command. After prompting for the user's current and new passwords, this client logs on to the server, checks out the appropriate user object, changes the password, and logs out. Clients could easily be written to do bulk edits to the database, or to tie automated processes of various sorts to the Ganymede server.

Most of our effort in client development has gone into developing a generic GUI client. This client has very little customization built in to it; it queries the server at run-time to get information about the schema being used. With a few exceptions to provide special handling for the mandatory object types that the Ganymede server depends on, it is completely generic, and can be used with any database definition in the server.

This primary Ganymede client displays a large window with two main panels. The left panel displays a tree that lists all the objects in the database, sorted by object type, while the right panel holds windows used for viewing and editing objects. The tree is built when a user first logs in, and can be used to browse the database. Each node of the tree has a context-sensitive menu associated with it, which is accessible by right-clicking on the node.

When the client first connects to the server, it opens a new transaction. The user can commit or cancel changes to the database made during the open transaction by using the "Commit" or "Cancel" button in the lower right-hand corner of the client. No changes are made to the database until the "Commit" button is clicked, so the user has the ability to cancel any actions he performed during the transaction.

Logging In

The client can be run either as an applet in a Web browser or as an application. When the Ganymede client is run in a Web browser, it appears as an applet in the browser's window, as shown in Figure 17. After entering a correct username and password, the

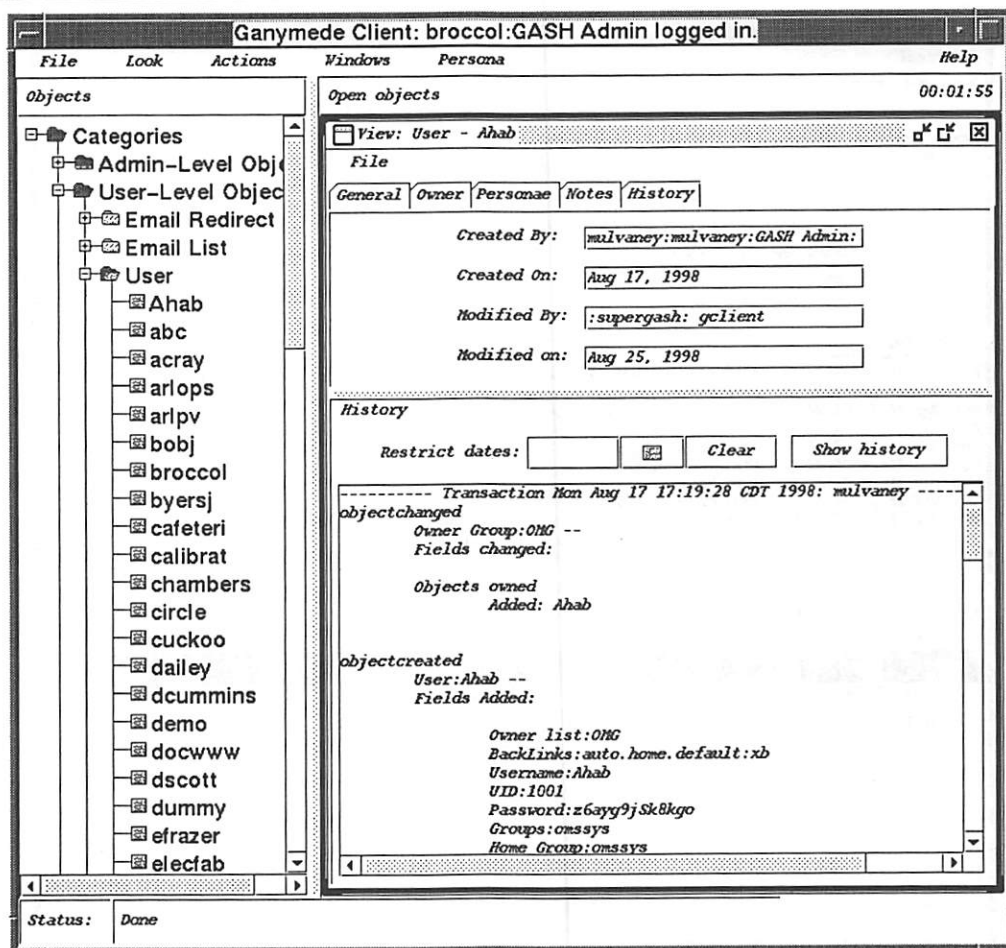


Figure 14: A User Object's Sordid History.

main client window appears. At this time, the client has queried the server to determine what sort of objects have been defined and to retrieve a definition for each type of object that the user will be able to edit. The client caches this information to speed up later operations.

When logging in as an end user, the only visible object in the tree is the user's own user object. In order to gain more privileges, a user can access his admin persona by choosing a persona on the **Persona** menu. The user will be presented with the dialog shown in **Figure 18**, where he will enter the password for his admin persona. Successfully changing to a new admin persona causes the client to rebuild the tree in order to reflect the expanded permissions, as well as to close any windows that might be open.

By default, the tree only shows those objects which are editable by the current admin persona, but the user can choose to have the tree display all the objects he has permission to view by selecting the **Show All Objects** menu item from the menu shown in **Figure 19**. Typically, the default role will grant users permission to view only a small number of objects,

such as other users in the same group. Most administrators will have roles assigned them which grant privilege to browse more of the database.

Editing Objects

After opening an object node, a list of the editable objects of that type is shown. When the "Edit Object" menu item is chosen from the tree's pop-up menu, an editable window is placed in the right-hand side panel. This window allows for the editing of this object.

When an object is first opened for editing or viewing, the server generates and sends the client a remote reference corresponding to that object. The client then talks directly to the object to determine what fields are defined within it, and builds up the fields displayed in the editing window.

Most types of fields in an object are displayed with simple GUI widgets: check boxes for boolean fields, date fields for dates, and text fields for single strings, IP addresses, and numbers. Object reference fields are either pull-down lists for scalar fields, or a composite selector for vector lists. The selector shows a list of available references on one side, and a list of

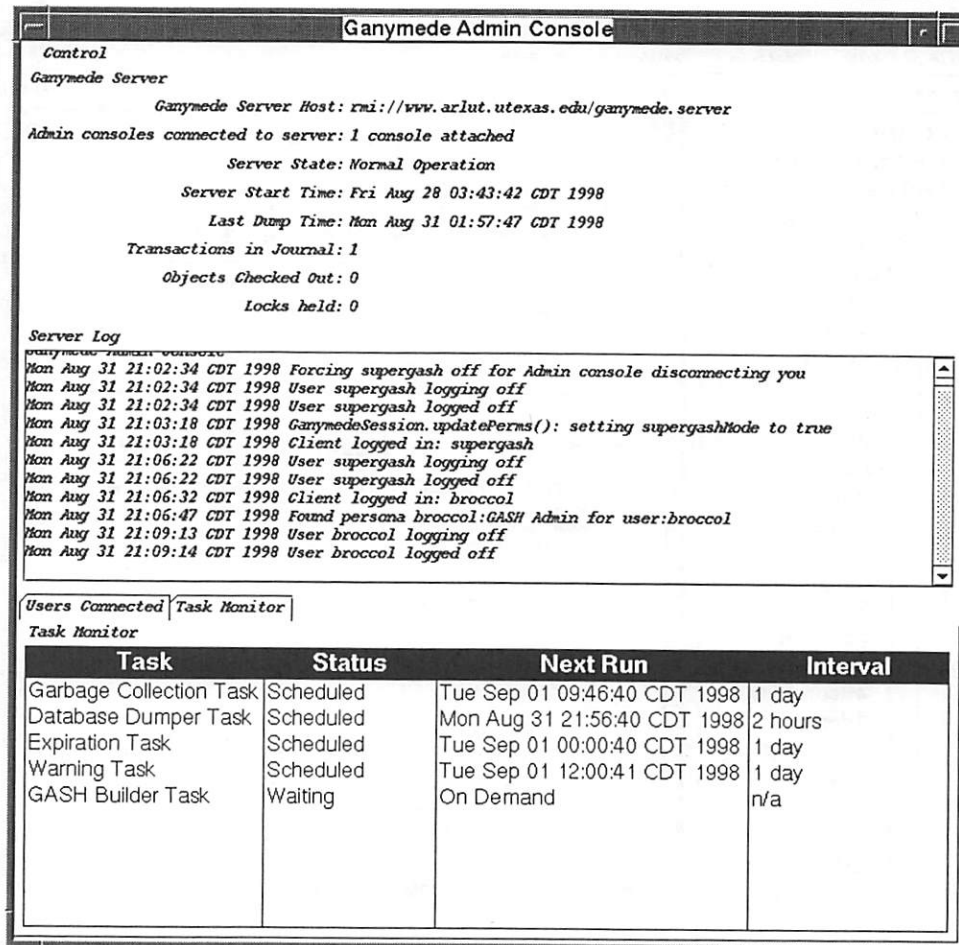


Figure 15: The Admin Console, Showing Registered Tasks.

selected references on the other. By moving the labels between the two lists, references are added or removed from the field.

Embedded objects are handled by embedding an edit panel within the main edit panel, as shown in

Figure 20. The embedded object's fields can be hidden or revealed by clicking on a icon in the panel's container. Because embedded objects are always employed in a vector context, there are also controls to add or delete objects of the appropriate type from the containing object. Embedded objects may themselves

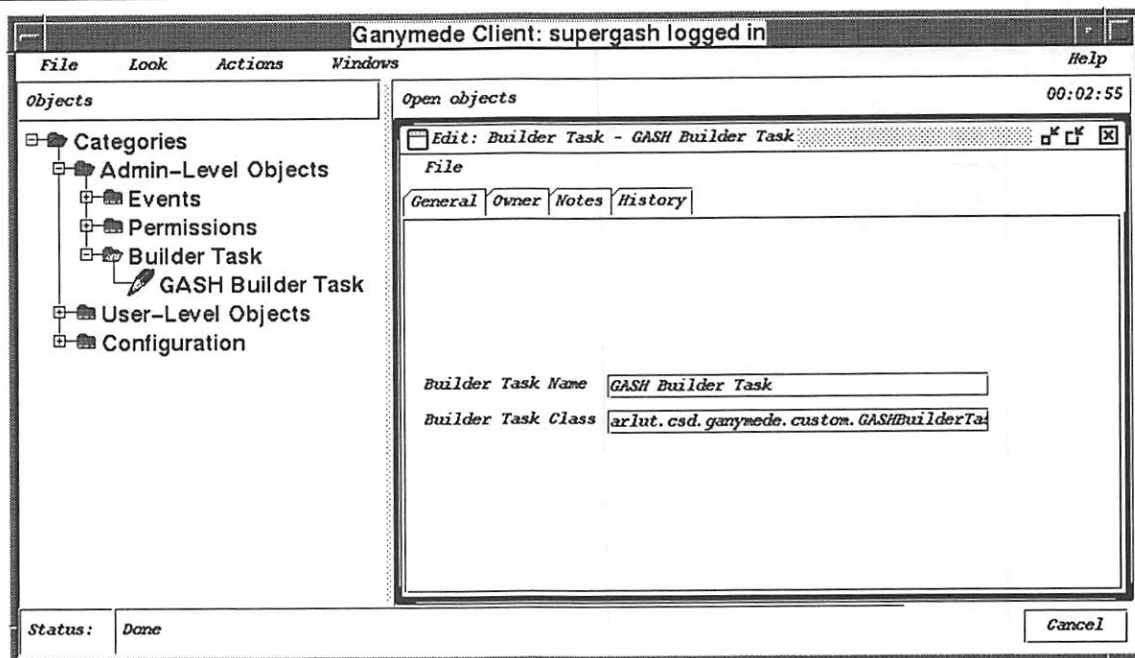


Figure 16: Registering a Builder Task.

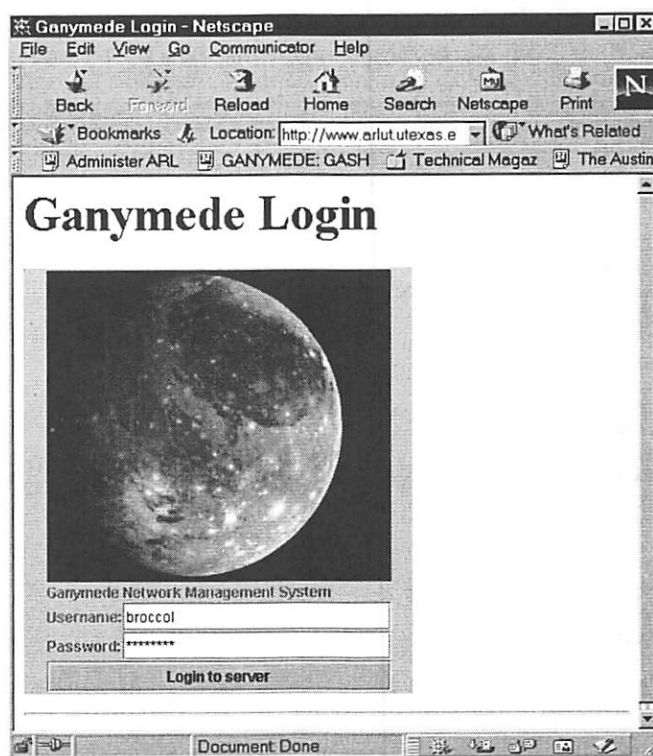


Figure 17: Ganymede Login Screen.

contain more embedded objects, and so a complex hierarchy of containment can be managed, if necessary.

Return Values

When a field is changed, the server reports to the client by sending a special object called a **return value**, which tells the client what happened as a result of the change. In a simple case, the return value

contains information about the success or failure of the change. The return value is capable of much more, however; it can instruct the client to display an informational dialog, and it can tell the client which other fields need to be rescanned because of the change to the current field, either in the current object or other open objects. This causes the client to query the server and update the specified fields immediately. Also, the server can use the dialog to initiate a

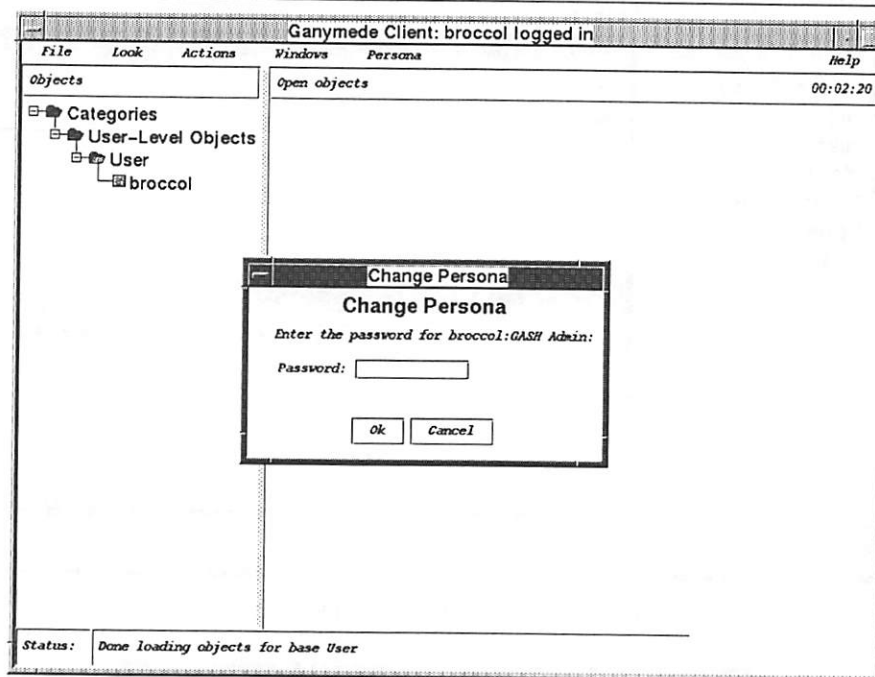


Figure 18: Admin Password Dialog.

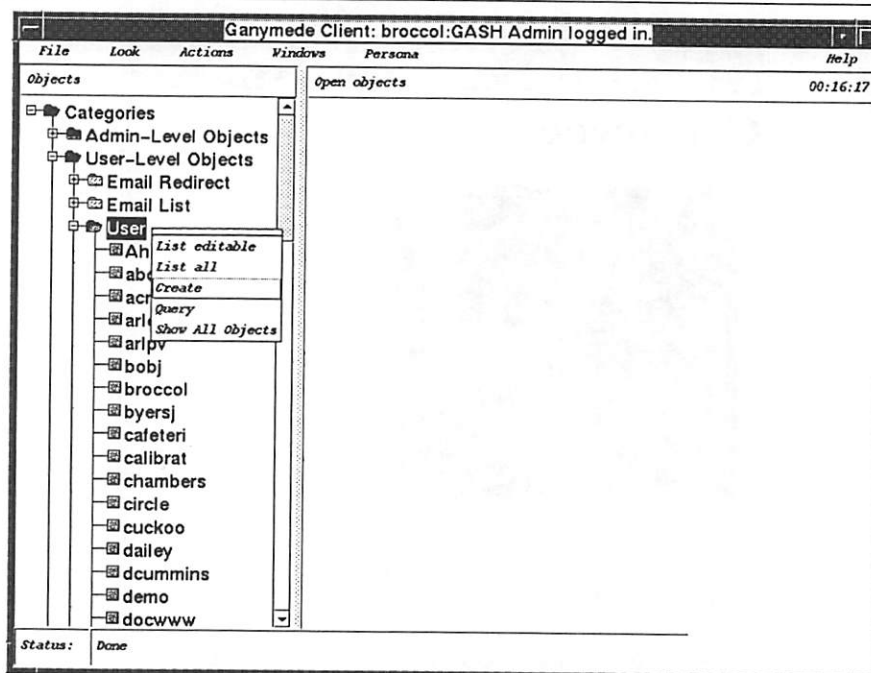


Figure 19: Object Menu.

complex series of wizards to gather more information about a complicated action.

Sometimes changes to the database require more information than can be provided by simply editing a

single field. In such cases, the server can send a definition for a custom dialog to the client. The dialog may include graphics, text, and a range of GUI fields. The information from this dialog is sent back to the

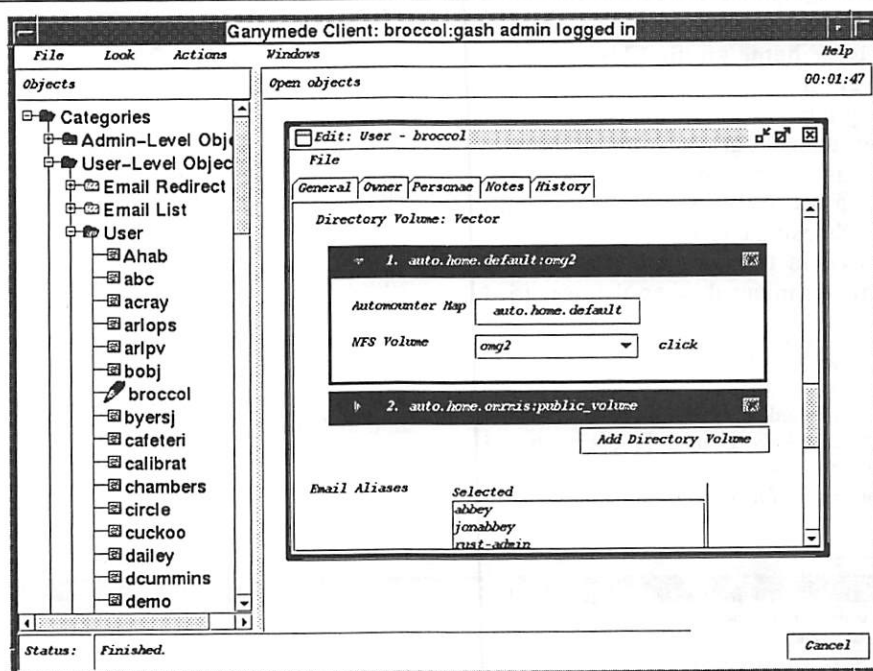


Figure 20: Editing an Embedded Object.

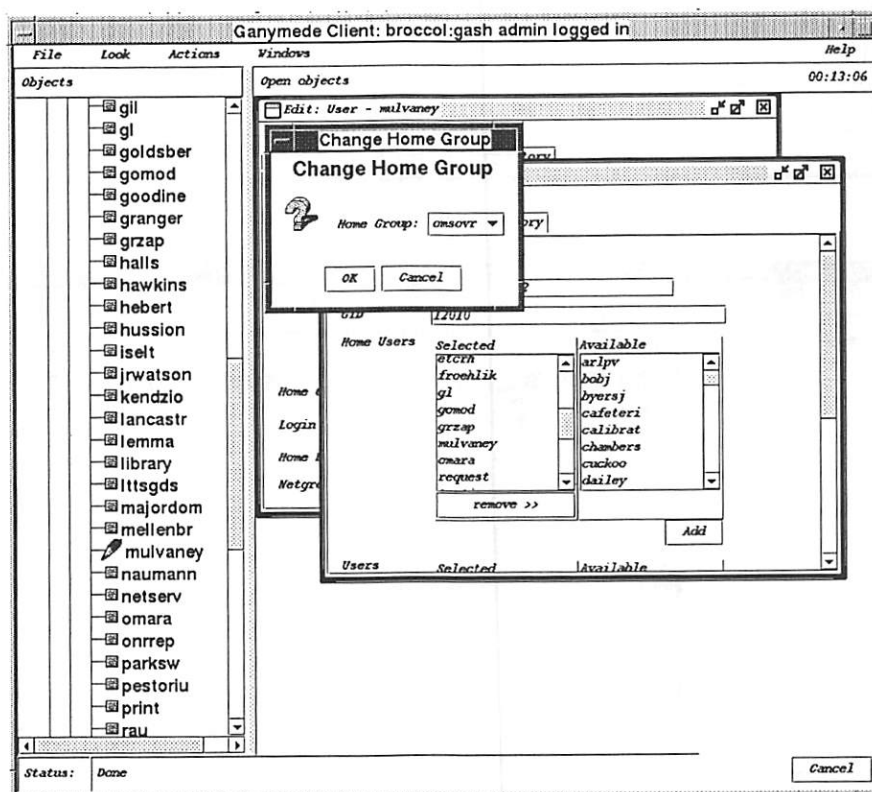


Figure 21: Picking a New Home Group.

server, which may in turn send down a new dialog to continue the discussion. In this way the server can walk the client through a series of steps using the traditional GUI wizard.

For example, when editing a group, removing a user reference from the **home users** field de-selects that group as the user's **home group**. This is demonstrated in **Figure 21**. In the GASH Schema, a user must have a home group, so a new home group must be chosen. In order to find out which group the user object should now have as the home group, the server sends a dialog with a list of the user's current groups to the client.³ After choosing a group from this list the server places the user in the new home group, and completes the operation initiated when the user tried

³Actually, the server only does this if the user belongs to two or more other groups. If the user only belongs to one group, that group is designated as the home group. If the user does not belong to any other groups, the operation will not be permitted. All of this logic is coded into the GASH schema, which we are using for demonstration purposes in this paper.

to modify the home users field.

Queries

The Ganymede server provides a powerful and flexible query mechanism. The client uses this system behind the scenes in many places. The client uses the server's generic query mechanism when it loads objects into the tree, for example. Users can create their own queries as well, using the **query box** shown in **Figure 22**.

The query box allows the user to build rather complex queries. The query box is interactive, and uses the client's cache of schema information to guide the user as to what fields and choices are available. Multiple query terms can be put together to specify as narrow a search as is desired. The user can further customize the query by indicating what fields should be returned in the server's report. When the query is submitted, the results are displayed in a table, like the one in **Figure 23**. The table contains one row for each object matching the query, and displays the values of the fields in the columns of the table. The table allows for sorting on each column, rearranging column

Figure 22: The Query Box.

Username	UID	Full Name	Groups	Home Group	Login Shell
backup	14045	Steve Unruh	spubak	spubak	/bin/tcsh
backup3	14064	Stephen Unruh	spubk3	spubk3	/bin/tcsh
barks	16797	Roger Barks	itddwi	itddwi	/bin/tcsh
barber	6230	Lewie Barber	asmail, asksq3	asmail	/bin/tcsh
bciff	8035	Cliff Vasicek	sgqtra, itdrfm	sgqtra	/bin/tcsh
be	10026	Hans Baade	evgads, evfsc, evfsc1, evfscn	evfsc1	/bin/tcsh
beal	6194	Travis Beal	asrmas	asrmas	/bin/tcsh
beaty	14039	Dotie Beaty	adgexe, spmail, adgfoc	spmail	/bin/tcsh
beck	8102	Matthew Beck	asrsom, asrabm	asrsom	/bin/tcsh
becker	8045	Barbara Becker	sgpoff, sgaadm, adgexe, sgmail, sgpb2g, adgfoc	sgpoff	/bin/tcsh
bellis	8052	Rebecca Ellis	sgpoff, sgaadm, adgexe, adgfoc	sgpoff	/bin/tcsh
benedict	8013	Sarah A. Benedict	sganim, sgwww, sgpdma, sgpfm, sgacom	sgpdma	/bin/tcsh
beth	6155	Beth Jones	asmail, adgexe, adgfoc	asmail	/bin/tcsh
bj	10031	Jack Shooter	evgads, evfsc1, evbfe, evtaap, evmdef	evgads	/bin/tcsh
bigatlin	4028	Betty Gatlin	adnadm, adgexe, adgfoc	adnadm	/bin/tcsh

Figure 23: The Query Result.

widths, and exporting the contents of the table either to disk or as an email message. Reports can be generated, either as tab- or comma-separated ASCII, or encoded as an HTML table.

Adoption Considerations

We hope that Ganymede will prove attractive to a large community of administrators. We feel that its sophistication, flexibility, and portability will allow it to be widely adopted in a way that GASH could not be. Ganymede is not appropriate for all environments, however. There are issues regarding security and performance that should be taken into account when considering Ganymede's suitability.

Security

While Ganymede has an effective permissions and ownership model, it does not support encrypted communications between the client and the server. At the current time, Ganymede is based on RMI over a non-encrypted socket layer, and as such is not appropriate for use on the open Internet for those who are concerned about packet-sniffing.

Sun is moving to support RMI over custom socket classes in the Java 1.2 JDK, which will make it possible to produce a version of Ganymede that uses Secure Socket Layer (SSL) encryption. Sun has not yet committed to producing a freely available implementation of SSL sockets for Java, but third-party companies have demonstrated RMI over SSL using their SSL implementations [8]. These implementations tend to be very pricey, however, and there are the inevitable U.S. export and licensing issues to consider.

When and if an affordable SSL solution for RMI appears, Ganymede could be retrofitted to provide a higher level of security with very little work.

Performance and Scalability

When we originally designed Ganymede, we set as a goal the ability for Ganymede to scale up to handle 50,000 network object records, a figure some six times higher than we were handling with GASH. The main factors determining how well Ganymede will scale are memory, Java Virtual Machine (JVM) efficiency, and thread and database contention in the Ganymede server.

Memory

Ganymede has limitations on its scalability due to its RAM-resident database. Our current database is loaded with DNS information for 2,087 systems with 2176 total interfaces, 48 administrators, 753 user accounts, 238 account groups, 1200 email alias records, 234 NIS netgroups, 520 NFS volume definitions, 753 automounter entries for home directories, and network connectivity information for 517 rooms. This comprises a total of about a megabyte of GASH data files in text form, some 8,526 objects or so in the Ganymede server.

With this data loaded, the Ganymede server takes up just under 40 megabytes of RAM. Since the machine we are running the Ganymede server on has a gigabyte of memory, scaling up by a factor of six or so would not be a problem for us, at least as far as absolute RAM consumption is concerned.

JVM Efficiency

The biggest problem with such scaling is not the actual consumption of memory, but the time required for the server to handle garbage collection. In an earlier stage of development, the Ganymede server was taking up nearly 100 megabytes of RAM with our dataset loaded, and we would experience frequent and significant lags in the server's responsiveness. We were, however, running the Ganymede server on a relatively slow, relatively loaded machine, a 60-MHz multiprocessor Sparc system, which was also responsible for the laboratory's Web, news, mail and file services. In addition, we were running the JVM in interpreted mode with debug logging enabled, rather than using the JVM's Just-In-Time compiler (JIT) to convert the Java code to Sparc machine code. The latest 1.2 beta seems to be much more efficient, however. We are expecting that a good generational garbage collector, such as is promised with Sun's next generation HotSpot JVM, will do away with concerns over garbage collection overhead, even with a substantially larger database.

The biggest delays that users are likely to encounter will occur when the builder tasks are running. The builder task base class is designed so that the builder task can lock the database while it assembles the data needed for the build, and then release it while it is running external scripts to process the data. While the first phase of the builder task is running, no transactions may proceed to commit. A common source of delay is for a user to make some changes, commit the transaction, then immediately try to make some more changes and commit that transaction. The second transaction cannot proceed to completion while the server is busy writing out data files in response to the first.

Thread and Database Contention

Which brings us to the issue of thread and database contention. The Ganymede server has been designed to try to keep thread contention low. Queries on the Ganymede database are a very frequent occurrence, and the query system has been specially optimized. Any number of clients can issue queries on the database without experiencing thread contention for access to the objects in the database, as long as no transactions are committing. While a transaction is being committed, no queries can be issued on object types involved in the transaction until the transaction finishes committing. Individual object accesses proceed normally, as the server does do table-level synchronization at all times. Queries are guaranteed to be

transaction-consistent, and are not processed while a transaction is being committed.

One important thing to note with regards to scalability is that the Ganymede server does not support multiple users simultaneously making changes to an individual object. This is most important with owner groups. If two administrators in the same owner group try to create an object and place it in that owner group, one of them will be unable to do so, and must wait until the other either commits or aborts his transaction so that the owner group is released. Whenever an object is added to or removed from an owner group, such contention can arise. Simply editing an object owned by a particular owner group, however, will not cause such contention.

The answer to this contention problem is to take advantage of the ability of the Ganymede server to support owner group hierarchies. If a group of admins gets large enough that the admins are often getting in each other's way trying to create or delete objects, it is a simple thing to create sub-owner groups that the objects can be placed in without contention.

Current Performance

With the current 1.2 beta 4 JDK we have had five users and an admin console connected to the Ganymede server simultaneously, with each user making changes and browsing the database without noticeable pauses. We do look forward to moving Ganymede to a modern UltraSparc server at some point, but even at the current level of performance, Ganymede provides very acceptable performance for our needs. Right now, we have less than 50 administrators registered in GASH, so it is rather unlikely that more than five admins will ever be using Ganymede at the same time. Once we have fully replaced GASH with Ganymede in the lab, we will be looking to see how well Ganymede scales. Ultimately, we hope to allow our end-users to have free access to Ganymede to handle their own passwords, shell information, and the like. At this time we don't know how heavily Ganymede will wind up being loaded in that environment, but it seems unlikely that too great a number of users would ever try to change their passwords at the same time.

Reflections on Java

When we first started investigating the possibility of developing a next-generation GASH, Java was far less developed than it is today. Version 1.0 was the exciting new thing, and critical features like RMI had not yet been released. But the promise of a sophisticated, operating-system agnostic development platform with widespread industry support was compelling. When Sun released RMI Alpha2 in mid-1996, it became clear that a distributed object design for Ganymede would be possible with Java. At that point, the basic outline of the Ganymede design started to take shape.

Since then, we have developed Ganymede as Java has itself been under development. In some cases we have wound up developing pieces that were not yet available from Sun, such as the tree and table components used in the client and the admin console. In many other cases, such as with RMI, we have found Sun providing just the right thing at the right time to make our work possible. We've had to wrestle with frustrating bugs as Java has matured, but the bugs got fixed. The momentum behind Java, both from Sun and from other companies has continually reinforced the appropriateness of our initial decision to go with Java.

Indeed, we do not believe that we could have done Ganymede with any other technology, given the resource constraints we were under. Java gave us a portable GUI, a distributed object API, a large set of thread-safe class libraries, and both memory and type safety. Most of these things can be had for C or C++, but only for serious money and/or limited portability. We certainly couldn't think of making a freely distributable tool using commercial C++ class libraries. With Java, we were able to write a distributed, multi-threaded, portable GUI application of 140,000 lines of code using nothing but X-Emacs and the Java Development Kit.

Concluding Thoughts

Ganymede represents an attempt to provide a comprehensive and flexible management system that can be placed on top of the existing directory infrastructure already in place in typical UNIX networks. It is not designed to answer all the questions about how to organize directory services, but rather to allow adopters to bring their own experience and environment into the design of their directory management tool box. In this respect it differs greatly from GASH, which had a particular network management philosophy hard-wired into its implementation.

Ganymede seems most similar to Novell's Novell Directory Services [9] and Microsoft's forthcoming Active Directory [10] in as much as it provides both a customizable directory database and a set of GUI tools to manage the database. It differs from these in that it does less; the Ganymede server is not designed to act as a high volume directory server. Ganymede does not support database replication or distributed management with multiple Ganymede servers. Instead it is designed to leverage existing directory mechanisms such as NIS, DNS, and commercial LDAP servers which have their own mechanisms for providing reliability and scalability through backup servers. Ganymede depends on the flexibility of scripting mechanisms on UNIX to provide support for getting the directory data where it needs to go. Finally, Ganymede does not provide native support for LDAP or, indeed, any other standard directory API. Ganymede can feed data to servers which support such API's, but administrative programs written to

manage directory services using such API's will find Ganymede, on the whole, an incompatible partner.

Ganymede does, however, provide a good solution for the medium-to-large-sized UNIX or mixed network. It has particularly good support for group administration, with the permissions system and the mail and logging system designed to facilitate administration teams. In addition, the Ganymede server is customizable in a rather deep way. Not only can the definition of the database schema be customized, but also a considerable amount of intelligence can be placed in the server using plug-in Java classes. This notion of an intelligent server is in keeping with our original design goal for GASH to produce a tool that would make it possible for a broad audience to safely manipulate our centralized directory information. Ganymede can be taught about a particular environment, and will work to keep it in good order.

Anticipated Future Developments

Barring some continuing polishing work, the Ganymede system is complete and ready to use as it is today. Our main goals at this point are to get Ganymede fully implemented within our laboratory, and to get Ganymede documented well enough that people can begin to work on developing their own custom schema kits. Essentially, most things that we can see needing to be done with Ganymede revolve around the development and elaboration of schema kits.

One important development would be to implement secure authentication and encryption for client-server communications. As we mentioned in our security discussion above, this will depend on an affordable implementation of the SSL protocol for Java.

One possible downside of Ganymede compared to GASH is that the Ganymede client requires a GUI display, whereas GASH could be run from a TTY console. It would be nice to have a completely functional and generic text client for Ganymede for those cases when a GUI display is not available.

Availability

Ganymede is currently available in pre-release at <ftp://ftp.arlut.utexas.edu/pub/ganymede/>, with documentation, screen shots, and information on joining the Ganymede developer's mailing list at <http://www.arlut.utexas.edu/gash2/>. While we don't yet have the licensing finalized, we intend to place Ganymede under the GNU Public License. Once we have Ganymede fully implemented in the laboratory and have licensing approved, we will formally release Ganymede 1.0.

Credits

Many people have contributed to the development of Ganymede. In addition to Jon and Mike, Navin Manohar and Erik Grostic made important

contributions to the client-side code development. Gil Kloepper provided design guidance for handling network issues and code for the GASH kit's back-end DNS support. Dan Scott supported the project administratively, and contributed greatly to the higher level design issues in Ganymede, as well as providing invaluable user interface design feedback and bug reporting.

A lot of Ganymede is based on the experience and design work that went into GASH. In addition to the aforementioned names, Dean Kennedy and Pug Bainter should be credited for their design work on GASH. Pug Bainter authored the original GASH makefiles that Ganymede's GASH kit uses to propagate information from Ganymede into NIS and DNS.

Finally, thanks to all of the administrators at ARL:UT and elsewhere who provided feedback on GASH and let it be known that things could perhaps be just a little bit better.

Author Information

Jonathan Abbey initiated the Ganymede project at ARL:UT in late 1995 and has been working on it pretty much nonstop ever since. He graduated with a B.S. in Computer Science in 1993 from The University of Texas at Austin, and has worked at Applied Research Laboratories since September 1989. He is reachable at jonabbey@arlut.utexas.edu.

Michael Mulvaney has worked on the Ganymede project since joining ARL:UT in early 1997, developing the Ganymede client. He graduated with a B.A. in Economics in 1996 from The University of Texas at Austin. He is reachable at mikem@mail.utexas.edu.

References

- [1] Abbey, J. "The Group Administration Shell and the GASH Network Computing Environment," *Proc. LISA VIII*, September, 1994.
- [2] Sun Microsystems, <http://java.sun.com/products/jdk/rmi/>.
- [3] Stern, H. *Managing NFS and NIS*, O'Reilly & Associates, Inc., 1991.
- [4] Albitz, P., Liu, C. *DNS and BIND*, O'Reilly & Associates, Inc., 1993.
- [5] Loomis, Mary E. S., *Object Databases: The Essentials*, Addison Wesley, 1995.
- [6] Hogan, C., Cox, A., Hunter, T. "Decentralising Distributed Systems Administration," *Proc. LISA IX*, September, 1995.
- [7] Object Management Group, <http://www.omg.org/>.
- [8] <http://www.java.sun.com/products/jdk/1.2/docs/guide/rmi/SSLInfo.html>.
- [9] Novell Corp., <http://www.novell.com/products/nds/index.html>.
- [10] Microsoft Corp., <http://www.microsoft.com/ntserver/basics/future/activedirectory/>.

Building An Enterprise Printing System

Ben Woodard – Cisco Systems

ABSTRACT

Cisco Systems has chosen to internally develop an enterprise wide print system that provides access to more than 2000 printers for both Unix and PCs. The requirements for this print system were that it had to be very cheap to construct, highly scalable, easily maintained by a very small staff, fault tolerant, and mission critical reliable. In other words, management essentially wanted everything for practically nothing. To meet our objectives we built our print system out of interchangeable low cost running PCs Linux, LPD and Samba as well as other standard Unix applications. The low cost of PC hardware and the lack of licensing fees for Linux allowed us to deploy the print system vary widely without having to go through all the managerial justifications necessary to authorize larger scale purchases. By making each print server interchangeable we achieved scalability as well as a certain degree of fault tolerance. The flexibility of running a Unix like operating system such as Linux as opposed to another more restrictive operating system allowed us to develop a worldwide printing application that can be managed very easily by only two or three people. And finally the robustness of Linux made it possible for us to use our print system in mission critical environments such as manufacturing production floors.

This paper discusses the process by which the print system was implemented and the wisdom learned in the process. It covers topics such as how to gain and maintain control of the printing process, why it is necessary and how to keep printers a completely network managed device, how we learned to deal with large numbers of server, the advantages and problems we ran into as the number of servers grew, and the many advantages and few disadvantages of basing the system entirely on free software. It also highlights some of the major processes that we automated and the success we had devolving power first to the local technical support people and then ultimately to the users. Finally, it discusses many of the problems that we are running into now that the print system is a few years old and the steps that we are taking to ensure that we do not become victims of our own success and that we do not have the whole system collapse due to data rot.

Since the real key to managing thousands of printers effectively is figuring out how to save time, the real world experience we gained and the time saving tips we discovered while learning how to manage thousands of printers should be valuable even to sysadmins that have only a few printers to manage.

Introduction

Managing just one printer can be difficult and just a few dozen could be enough to keep a full time sysadmin perpetually busy. When the problem is scaled 1000 or 2000 times to the size of the enterprise for a medium sized multinational corporation, the problem has the potential to be an unmanageable nightmare. We have heard vendors pitching printing software use figures such as 70% of the time spent by a network administrator is spent resolving printer related issues. This paper discusses how we were able to build a greater than 2000 printer print system for Unix workstations and servers as well as PCs that is managed by a staff of only two full time sysadmin. Since the key to managing thousands of printers effectively is figuring out how to save time, the time saving tips we discovered while learning how to manage thousands of printers should be applicable to sysadmins that have only a few printers to manage.

Getting Control ...

... of Unix

When my colleague, Damian,¹ arrived on the job there was no printing system to speak of. There were a couple of print servers but hardly anyone used them. The job of the person in charge of printing was basically to keep all the Unix servers' printcap files up to date. This was a monotonous job and he quickly tired of it. The first thing that he did was generate a canonical list of all the printers that he knew about and then sift through this list to find the ones that actually worked. This winnowing process can be greatly sped up today since most printers support protocols such as SNMP. Armed with the authoritative list of all the

¹My colleague Damian Ivereigh actually began work on the print system in June of 1995. I joined the project around October of 1996. To contact Damian you can send email to damian@cisco.com.

printers, he made a script to distribute this to all of the Unix servers. In talking to several people, this beginning is where their print system ends. This in itself became a time consuming and cumbersome task but at least all of the Unix servers could print to all of the printers.

Keeping up to date with all the hosts that needed the printcap file quickly became a hassle and the repetition was boring him and so he decided to fix this problem once and for all. The key observation was that there were basically two printcap files. The one for the print server which sent the jobs to the printers and the ones that went on the Unix print clients which sent the print jobs to the print servers. The printcap file on the Unix servers that were not print servers was very repetitious with entry after entry redirecting the print jobs to the print server. The non print servers really didn't need to know anything about any printer except for its queue name and the name of the print server. Damian came up with a way to get out of all of this repetitious work. He got the source for LPR and modified it so that it would read a file called `"/etc/lpr.conf"` instead of `"/etc/printcap"`. In this file, he put all the things that were common in every printcap entry, the name of the print server, and the root of the spool directory. Then he modified the routine in the LPR source code that found the printcap entry so that it would create the spool directory if necessary and generate a simple printcap entry that would redirect the print job to the print server. This way every time any of the LPD utilities looked up a printer, it would find an entry that simply directed the job to the print server.

This was a great boon. It made it so that once he installed all his modified print clients he could basically forget about the existence of the machine. It would have access to every printer that the print servers knew about. Another innovation was to wrap this up with a very quick and easy installation procedure. He made all of the print software into a shar file. So all that a sysadmin had to do was simply run this shar file and the script would uudecode the tar file that had the modified binaries for that architecture, put them in place, and restart the print services on that machine. Reducing the entire task of getting and keeping printing running on your machine down to the simple process of fpt'ing a file and running it as root almost instantly won the unanimous support of all the other sysadmins. It made it so easy to go along with the system, that almost no management leverage was needed to get several groups of sysadmins to install this print client. In an organization such as ours where we have a relatively large number of sysadmin arranged into several almost autonomous groups, getting such widespread acceptance was no small feat.

Every once in a great while we are asked to come up with a new port of our print software, there was one patch release to fix a bug, and every so often a new vendor patch will overwrite the software but

other than that, the Unix LPR client has needed almost no maintenance. All the sysadmins know that part of process of setting up a new machine or upgrading the OS on a machine is to download and run our print software installer.

... of PC's

About this time, early in 1996, PCs were starting to replace the Macs at Cisco. It quickly became apparent that there was a lot of duplication between the team that was keeping the printers up to date on the NT servers and the person that kept the printers running for the Unix servers. The NT folk were already fairly busy trying to just keep their systems up and running and didn't want to bother with printing and so Damian fired up Samba on the backup print server to provide PC printing.

The PC population was even easier to convince to use the print servers than the Unix folk. They really didn't have much choice, it was generally technically beyond the majority them to install programs such as JetAdmin, or WLprSpool and print directly to the printer.

Drivers were a perpetual problem until we bought Jeremy Alison, one of the Samba developers, enough beer that he came over and got the Plug and Print automatic driver download for Windows 95 working for us. Now when a Windows 95 client double clicks on a printer on the print server for the first time it silently downloads the correct driver to their PC. This has many benefits. First of all, it reduces support calls by allowing us to make sure that the client is using the most up to date, most bug free driver that we can find. It eliminates the driver/device mismatch problems that can complicate printing. It also minimizes the amount of time that our internal technical support staff (or sometimes us) has to spend on the phone with a user to get a printer set up. We currently can't pull the same trick with NT servers because Samba doesn't implement the NT RPCs yet but we have found a place in the registry where NT looks to find the print drivers. This allows us to place all the needed drivers on the NT workstation's disk and have it load the correct driver from its disk much like it would have done had it downloaded it off of the print server. The samba developers assure us they are working on a plug and print implementation that works with NT and that it will be done soon.

Taking control of the PC printing as well as the Unix printing is quite a time saver for the organization. Instead of two groups trying to maintain an accurate list of the printers, you only have one and when you fix a printing problem with one, most frequently, you have also fixed the problem with the other. In other organizations, we have heard of many cases where the PC's have one printer and the Unix machines have another printer and each printer is fairly underutilized. We have also heard of cases where the PC support guy fixes a printer for the PC's

and in so doing breaks the printer for the Unix machines. Adding the appropriate entry to the `smb.conf` file was just another step in setting a printer up.

... of Printers

Another step in our complete take over of the print infrastructure was taking complete control over the printers. The first thing that we took control of was the printer standards. There were no IS enforced standards for printers at the time when Damian started. This led to some very unfortunate purchases in the early years. We have pretty much weeded all of the bad printers out but it took quite a while and quite a bit of coercion. The key thing for a printer in a large enterprise environment is that it must have a **great** network interface. You have to keep in mind that the network is your only connection to the printer and if you have trouble communicating with the printer or you can't do all jobs you will ever need to do or find out all the information that you might ever need to find out from the network interface, then you are stuck. Attention to detail is critically important when evaluating printers; it is rather amazing how very simple seemingly harmless semantic changes can cause problems in an enterprise. Another thing to keep in mind is that even simple tasks can become quite time consuming when they are multiplied across more than 1000 printers. Therefore everything must be scriptable or you cannot automate the job. In looking through many printers' network interfaces with a fine toothed comb, we have found that there are only a couple that really can stand up to the demands of enterprise printing: HP, Lexmark, and Tektronix, though Xerox is starting to come along.

In evaluating a printer's network interface don't be misled by the printer vendor's cool GUI program. Unfortunately, the current trend with the printer vendors is to provide a very dumbed down graphical user interface that allows you to do "everything that you will ever need to do" with the printer. The truth is that graphical user interfaces are practically useless for more than a handful of printers. Intrinsic in these GUIs the developers have made a bunch of insidious assumptions about how you want to view and work with the data. You are frequently stuck when you try to do something out of the ordinary. The biggest problem is that they are not scriptable. This is why we have resorted to using some of the more low level protocols such as SNMP for dealing with printers. In fact most of those fancy, command line and GUI tools are just wrappers on simple SNMP calls. When evaluating a printer, we first look at what the GUI tools can do, and make note of any interesting new features. Then we throw them away and never look at them again because they just can't be used in an environment as large as ours. Even the command line tools such as `hpnadmin` don't generate the kind of output that lends itself easy to processing in a shell script. We were forced to write many of our own tools. Hopefully

by the time that you read this, many of these tools that we have developed in house will be available on the Internet.² The next aspect of a printer's life that you want to control is its bootup sequence. There are quite a large number of reasons that you want to control the bootup sequence of the printer. The first reason is that you need to know what IP address the printer. The second reason is it allows you to do some simple configuration by using a bootp configuration file. Both HP and Lexmarks have a bootp configuration file. The details differ but they allow a few simple configuration parameters such as setting the community name and access lists. Access lists are remarkably useful items; they allow you to limit access to the printer to only the print server. This makes it so that you have a single point of queuing. This is a major benefit to technical support staff because if someone says that they are having trouble with a print job stuck in the print queue for this and such printer, they know that the job is stuck in the one and only print queue for that printer. They don't have to search for which queue it is in. It also gives them one place to troubleshoot printer problems and one place to delete problem print jobs. This is a great time saver for them as well as us.

Another reason that you want to control the boot process is it gives you a chance to do some post boot configuration. Both HP and Lexmark TFTP their bootp configuration file. By using tcp wrappers and having the tftp of the boot file logged and having a program continuously looking at this file you can tell when a printer is booted up. You simply wait some appropriate amount of time and then use whatever tools you have to configure the printers fully. You can also use a SNMP trap to figure out when the printer is booted up. We basically use our SNMP tools to disable unused protocols, check the firmware version etc. The value of this post boot configuration should not be overlooked. It allows you to bring the configuration of all of the printers into harmony and therefor accumulate within the configuration of the printer all the wisdom that you have learned by configuring all the other printers of the same type. One of the things that I have learned managing a large environment is that NVRAM is volatile. You cannot count on the fact that you configured this printer once when it was first set up and so all the appropriate settings are still set.

When we first started the print system many of the printers were configured by either the front panel or telnet which caused problems when they were moved or the IP address range for the local LAN was changed. Someone had to physically go to the printer and change it. When you have a printer configured with a static IP address, if anything changes such as the subnet it is on or the IP address range of the network, the printer quickly ceases being a network managed device. Some of the newer printers grabbed

²<http://pasta.penguincomputing.com/pub/prtools>

DHCP IP addresses which also proved problematic. There are several problems with using DHCP for printers. The first is that the print system is not informed when a printer changes its subnet or its IP address changes. The second problem is that the print system is not notified when the printer boots up and so you do cannot provide a boot configuration file nor can you do any post boot configuration. HP is currently addressing this problem with the latest version of firmware in the JetDirect EIO cards. These new cards send out a multicast packet when they boot up which can be monitored by the print system to initiate all sorts of post boot configuration.

We resorted to using bootp to configure our printers. With the current technology, bootp is probably the best method to configure a printer. Adding the printer to the bootptab became just another step in the process of setting a printer up. The main advantage of bootp is that the printer always remains a network managed device. The moment a printer ceases to be network managed and you or someone else has to go out and visit a printer, you have lost a great deal of time. The second benefit of running printers off of bootp is that you can specify a configuration file for them to tftp down after they boot up. Using TFTP as a trigger allows you do all sorts of post boot configuration of the printer. There are two main down sides to using BOOTP. The first is if a printer moves subnets, then you must modify the bootptab. The second is that you have to know what devices you are going to BOOTP. We overcame both of these limitations by modifying the bootpd. What we did was made it so that when a device such as a printer sent out a BOOTP request, we checked to make sure that it was coming from the network we expected it to come from. If it did not we quickly rewrote its bootptab entry so that it had an appropriate IP address and informed the other aspects of the print system that needed to know about this change. This allowed us to get out of the moves process which we greatly despised because it was always interrupting our weekends. The second modification we made BOOTP seem even more like DHCP. If we get a BOOTP request from a device we had never heard of before, we allocate an address for it and set it up in the print system with a dummy name. That way when a user calls us asking us to set up a printer, it is largely already done, all we have to do is rename the printer. This is an incredible time saver.³

Automate Everything

With the rapid growth of Cisco (and therefore our workload) coupled with the complete lack of growth of our staff, we were forced to automate everything. Other than expanding responsibilities, with no corresponding growth of resources, there are many good reasons to automate every aspect of system

management. The first reason is that in many places you have the same information. Making sure that all these areas are kept up to date with each other can be difficult and the problems associated with having one piece of information out of sync with another can be difficult to track down. As a rule you want to put information one place and then have the computer copy it to all the places it needs to be. A good example is the printer's IP address. Several parts of the print system need to know the printer's IP address. For example the filter scripts need to know the printer's IP address so that they know where to send the print data, the BOOTP daemon needs to know what IP address to give the printer when it boots up and the several scripts that we use to help administer the print system need to know the printer's IP address so that it can do all the SNMP queries to find out the status of the printer. You don't want to have to maintain the IP address of the printer in three places. So what we initially did was put all this information in a flat file and then wrote programs to read this flat file and turn it into the needed configuration files such as `/etc/printcap`, `/etc/bootptab`, and `/etc/smb.conf`. Now we have progressed beyond that and we store the printer information in a little home grown directory service. We have also modified certain critical programs such as `bootpd` and `lpd` so that they read directly from the directory service.

I have already discussed some of the first tasks we automated: setting up the spool directories, building of the `"/etc/printcap"`, the `"/etc/bootptab"`, and the `"/etc/smb.conf"`. The next big time saver was the modifications to `bootpd` which automatically set up or moved printers.

DNS and Oracle

A couple other chores that we automated were keeping DNS and the Oracle databases up to date with our printer database. Although both of these programs were great time savers, they both took three or four times as long as expected to get into production. Doing a post mortem on the project and why it took so long to implement, took us quite a while but led to a useful piece of insight. It turns out that the problem was that both of these programs reached into someone else's database and updated their records. In so doing, they pushed back the frontier of what is managed by the print team. The added complexity of dealing with someone else's system made these task more challenging and introduced some support cross functional support challenges. I have run into cases where a script that ran fine a couple of days ago quit working all of the sudden because some interface or some subtle semantic has changed. For this reason, we don't rely on any of these programs for mission critical functions. This allows us a couple of days to figure out what is going wrong with the script and fix it before it becomes a problem. We also plan accordingly when we begin a project that will involve working with another group.

³So much so that it even surprised us.

The Script-able Ultimatum

When evaluating printers the approach we took was that every task we would need to do on a regular basis needed to be automatable. If something required a step that we couldn't write into a script then we couldn't apply it. There were several features on certain printers which could only be set up through telnet or through some web interface, we deemed these features unusable. If they were in critical operations such as booting, then we deemed the printer unusable. There are quite a few operations that the vendors will tell you don't need to be script-able because "you will only have to do them once a year" like upgrading the firmware. In practice however, these operations come up more frequently than you would imagine. For example, say there is a nasty bug in the firmware of some printer, you have a fix for it but the printers that have been manufactured but not delivered yet might still have the old firmware. So for the next two months every printer that you set up needs a firmware upgrade. In Cisco, that translates to more than 90 printers. Repeating a simple operation 90 times can be really time consuming.

Automation and its effects on you

Managing a system this large is a demanding job both technically and psychologically. We discovered that we as the administrators of the print system are part of the print system. Our psychological state really affects the smooth operation of the system as a whole. We are the vision, the creativity, the wisdom and intelligence behind the print system and we are not as creative when we are feeling burned out. Managing a large print system can really quickly burn out any sysadmin if they are not careful. The thing that gets to you is the fact that you are dealing with literally hundreds or thousands of devices that have similar problems. The repetition can easily grind you down. That is one of the reasons that automating as many tasks as possible is so critically important. In most other companies that we have talked to this is really a problem. Printing is a job dumped onto the lowest man in the pecking order. As soon as that person can, he will pass responsibility onto another person. Consequently, there is no consistent vision or direction to the system and the wisdom gained by one generation is not passed onto the next. To maintain a reliable print system, it is vitally necessary to have that consistency of vision behind it. Print administration will kill your spirit very quickly if you do not automate it.

One problem that we had was the feeling that we were on a treadmill. Even with our reasonably successful automation of many processes sometimes we feel like things never change and never get better, like we are still plagued by the same problems that we had a year ago. Automation in itself feels like a never ending task that seems to have no noticeable benefit. What seems to happen is that the moment that we get one task automated, we have more time to address

some issues that were in the background. They end up being a can of worms and our time is once again completely used up. Print services can be maddening that way. The only thing that seems to help is trying really hard to maintain perspective. Knowing where we are, where we have been and where we are going. Every so often we take a look back at what we have accomplished and find that we are doing much more in less time. For example at Cisco, it used to be one man's job to take care of 180 printers, and two print servers for 30 some odd Unix servers. Now it is two man's jobs to take care of 2000 printers, close to 100 print servers, for more than 100 Unix servers, a few hundred Unix workstations and close to ten thousand PCs. Every little thing that we automate adds up to more time savings and helps push you back burnout just a little bit farther.

Power to the People

One of the greatest time savers that we put together was providing a web interface to the print system. We initially rolled this out to the internal technical support people and once we were sure that it worked acceptably, we rolled it out to the rest of our users. The web page provides users with the capability to start and stop print queues and delete print jobs. At first there was some apprehension to allowing every user to do this. In the more than two years that it has been available, we have yet to hear one report of someone misusing the privilege. One thing about the web interface to the print queue is that it is accessible to everybody and it allows people to solve their most common problems by themselves rather than calling us. We have found that this feature alone really eases the load on us and makes our customers much happier.

Dealing With More Servers

Originally we had two print servers, a primary and a backup. The primary was the one that did all the print spooling and the backup had been pressed into service for the Windows clients. This was a very simple system and worked fine for headquarters. Then our company decided to open a facility all the way across the country in Research Triangle Park, North Carolina (RTP). Since one of the purposes for this new site was disaster recovery and because some production servers there were going to need to print to printers at headquarters and vice versa, we were given a requirement that all the print servers need to be able to print to all the printers. This one decision more than anything else took us down the road that led to an enterprise print system rather than just maintaining local printers. Allowing all the server's in the world to print to all the printers without unnecessarily burdening the WAN introduces a couple new twists to the printing architecture.

First of all there was the matter of the printcap file. We had to make the scripts that made the printcap file know where the printer was in relation to the print

server whose printcap file was being generated. This introduced two new pieces of information into the print system. The first was the concept of a print server having a "location." The second was that a printer had a "location." For convenience we grouped the printers together into locations such as San Jose and RTP and then assigned a print server to a location. All the other print servers that were not spooling that particular location code still created the spool directories and had bootp records but in the printcap file they made the printer a remote printer. That way if a print job landed on any print server that was not spooling that particular printer, it would be redirected to the print server that was serving that printer. This effectively turned all the print servers into print job routers.

We didn't realize this at the time but this also allowed us to scale the print system almost infinitely where needed. By keeping the size of the location codes reasonably small and then assigning quite a few of them to a print server. We have been able to very easily add capacity where needed. When we need more capacity, we simply add a new server and give it some of the location codes. We currently have twenty-four print servers at our biggest site, headquarters. This scalability is one of the true successes of our print system. Cisco has been doubling its size almost every year for several years now and the print system has been able to keep up with this growth without having to spend a lot of money. In fact, we have been using desktop PCs running Linux as our print servers and so anytime we need to add capacity or place a print server at a new site, it is simply another \$1000 server. \$1000 is well within the purchase authority of even the lowest level manager and so it is comparatively easy to get approval. One of the problems that we ran into while adding capacity in this way was as the number of print server grew past six to ten, twelve, or twenty four, the job of dividing the locations amongst the print servers became more difficult. We eventually ended up writing a program to do it for us because the problem became so intractable. To make it easy for people to find printers and to make it so that we could easily divide the load amongst many servers we made the locations pretty fine grained, one floor of one building. As Cisco grew and the number of locations grew appropriately allocating these to servers became increasingly time consuming.

Another feature that dividing the print system into locations did for us was it allowed us to provide a more reliable service. If a print server was down for whatever reason, we simply moved the location codes to the rest of the servers. We found that it was difficult remembering which locations went where once the server came back up and we also found that our typical outage, a reboot, took less time than moving the location codes to a different machine. We greatly simplified this by giving each location a primary and a backup print server. That way if we marked a print server as down, the print jobs would go through to the

backup print server and we wouldn't have to move the locations. To make things even more reliable, if both the primary and the backup print server is down things are spooled from our "last chance server". To date, we have never had to use the last chance server. If we know that a print server is going to be down for more than a few minutes, we re-balance the load amongst the remaining print servers by running the same script that we use to fairly distribute the load amongst the servers. This resets the primary and the backup print server for each location without considering the print servers that are marked as down. Since we put these features in place, we have had essentially no down time. It has also made it so that we really don't have to come in on weekends anymore to do system maintenance. For instance, we have done things like one by one moved all the print servers from one server room to another server room during the working day without causing any user noticeable down time. The only downside to this is we are not collecting anywhere as much on-call pay as we once did.

Having print servers at multiple sites created one significant problem. We were using programs to read flat files and generate all the print server configuration files. When we only had one site, and one print server we were able to edit these files on this machine and then we would build the configuration for that print server right there. As we got more print servers we had to make sure that this file was up to date on all print servers. Initially we were using rcp to copy the file from the source to the outlying print servers. As the number of servers increased, it became a bit of a problem keeping track of which print servers had an up to date set of printer information files from which to build its configuration files. There was also the problem of multiple users. Since it was a simple file, we had to serialize access to this file so that changes were not lost when two people accidentally edited the file at the same time. We found that the local sysadmin had better knowledge of the local printers than we did back at headquarters and so it was helpful for them to make the changes that pertained to their sites. With the initial system of a couple of flat files, if the network was down between the sites, they couldn't make any changes to their print environment. All in all this was unacceptable. So we created what we thought was a database of the print information. (It eventually turned out to be more of a directory service like DNS rather than a database like Oracle in the end.) We wanted it to take care of the propagation of the printer information so that we wouldn't be tied to rcp and we wanted it to keep track of which servers had up to date information. We also wanted it to update a particular value in a record rather than locking the whole table. That way we didn't have to worry about serializing access to the table. We could allow people to update particular fields in a record and make that an atomic operation. We also wanted it to allow local people to update their print environment even if the network was down

and when the network was reconnected we wanted all the changes that they made to propagate around the world.

Damian who wrote this program called it SDDb for simple distributed data base. As it turned out this is quite a misnomer. First of all it is not simple; it is distributed, but it really isn't a database. Name notwithstanding this program is at the heart of our print system. We provide a front end for people to update records and all the programs (except for Samba) have been modified to read directly from the SDDb directory service rather than reading from their native files. This allows changes that we make to be propagated more quickly and also allows us to react more quickly to server failures or other interruptions in service.

With the rapid proliferation of servers to handle capacity and at remote sites, we ran into several problems keeping them up to date. The first problem we addressed was keeping our custom print software up to date on all the print servers worldwide. Although, it is not really an ideal solution, we use `rdist` to push out the binaries to the machines. We have not addressed the problems associated with servers that don't have the data pushed to them properly. As there are getting to be more servers, this is becoming a more and more of a problem. `rdist` just isn't designed to keep large numbers of hosts in sync with each other. Ultimately, we will probably move away from a push model and go to a pull model based upon RedHat Software's RPM. That way when the machines boot and every night, they go and get their configuration. If they happen to be down, when they next boot they will get their correct configuration. Or if the net is down when we happen to do an `rdist` to them, then instead of missing that update, they will simply get their correct configuration the next night.

Another problem we addressed was how to load up lots of machines quickly. When you are deploying two or three new servers per week, the time taken to load up a new machine is significant. What we did was created a NFS boot floppy that boots up the machine and then partitions the hard disk and loads all the initial software. We sort of use a basic RedHat 4.2 configuration. After the disk is partitioned and formatted, we lay down a skeleton directory tree and then just start dropping rpm's onto the disk. Once again this was an interim solution that hasn't been replaced yet.

One of the big problems that we need to address is although we have a way to make a brand new machine with all the patches applied, and we have a way to keep all our custom software up to date, we have yet to make a system where we can quickly and easily apply updates to all the print servers. At the moment, when in our opinion a machine gets too far out of date, we have someone put a floppy into it and then we dd the new install NFS boot floppy onto it and reboot the machine. It comes up and reloads itself using the exact process we use to create a new

machine. Once it is up and we are satisfied that it updated itself properly, we "zap" the floppy and put a new boot sector on it so that the system will boot to the kernel on the hard drive. Some time later someone comes along and removes the floppy for us. This can be a really time consuming task and we can only really do one machine at a time.

We currently are working on a solution to address all three problems as well as solve a few other problems we haven't yet addressed. We want a system that will create a new server from scratch over the net, will keep the custom software up to date, keep the vendor supplied patches up to date, allow us to have variations from our standard configuration, allow us to partially roll out test versions of our software and finally, allow us to boot off a special recovery partition of the disk which will allow us to rebuild the disk in the event of a bad crash that leave the main partitions unbootable. The way that we plan to tackle this problem is to have a kind of target configuration for each print server. Then when a server boots up, it will have a current configuration. If those do not match it will go through a series of steps that will turn one state into another state. This process will probably be done by using RPM controlled by make files. Hopefully, this will allow us to maintain even more servers with less effort.

Printer management

Since we have been running the print system for more than two years, we have accumulated quite a bit of data rot. For example, we have an automatic process that moves a printer within the print system if it comes up on a different network. We had a large site that moved and once everybody was out of the building there were six printers left. When we looked more carefully, we discovered that these printers had been decommissioned long ago but no process had been put into place to notify us. We still don't have such a process. Instead of trying to come up with a process to deal with this, a simpler solution is to leverage the work of another team who's specialty is monitoring servers and routers and other pieces of IS infrastructure and have them monitor the printers. Then if a printer drops off the network for more than a few days we can call the printer's contact and find out if the printer has been decommissioned or if it is broken or something like that. We can also use the the printer monitoring facility to find out many other pieces of information about the printers such as how well utilized our printers are. Are we over utilizing or under utilizing our printers? We also will be able to find out if certain models of printer are not standing up to the rigors of use.

Another kind of data rot we have is keeping the contacts and the locations of printers up to date. We try to set the location and contact when a printer is first set up but ever since we have automated the move process to the point where a printer can be moved

without intervention by us, we have found that the locations and the contact information for the printer is not being kept up to date. We have the beginnings of a system to keep the printer's contact name and location up to date. If someone prints to a printer that doesn't have the location and contact set, we will send email out to the user asking them for the information. Eventually, we find someone who gets fed up with getting email from us and decides to tell us where the printer is and who to contact regarding it. However, we need to check this information every so often and we have yet to put in place code that double checks the location and contact every so often.

Yet another kind of data rot that we are experiencing is with regards to names. It would be nice if a printer had one canonical name, however printers seem to accumulate names. These alternate names appear for various reasons. The first place is when a printer is renamed, we keep the old name around for a while to maintain backward compatibility. This problem was exacerbated by the fact that we used to name printers with a bit of their location in their name. For example, a printer in building K on the second floor might be called k2-opsteam. The problem is that with the rate that Cisco has been growing, groups are always being moved from building to building to make room for expansion. So every time a printer moves it would pick up another alias. We have no automated procedure to trim down the number of aliases a printer has. To try to curtail the explosion of names, we have stopped encoding the printer's location in the printer's name. That way a printer named opsteam doesn't have to change its name every time it moves.

One kind of data rot that we have taken very seriously is getting people to use our print system rather than the old NT server print services. When we take over a site that used to have an NT server in it, we redirect all the NT print queues to the print server. That way the local technical support does not have to make changes to everyone at that site. Then when the users print through the NT server, we send them an email telling them to change their printer configuration to point to the Linux print server rather than NT server. This eases the transition between the two services and makes it much easier for the local support people. They don't have to change all their clients' configuration; the clients will get email telling them how to do it.

There are certain places within the company on the fringes of IS support or in new acquisitions where we don't have an accurate list of all the printers. We need to put in place a process where we scan the network for unknown printers so that we can keep our list of available printers accurate and up to date. Right now this is a very time and network intensive process because you pretty much have to probe every address on the network. HP is addressing this problem by having their printers respond to multicast requests but it

will be a couple of years before this feature is widely deployed.

Open Source Software – Beyond religion

There is a lot of religion around the Open Software movement both sides have their reasons for sticking to their point of view. We brought free software and many of the ideas associated with it, into Cisco and in the beginning it was a bit of a struggle. When we started using Linux for a mission critical function such as printing was pretty much unheard of within a large company. Now it is fairly commonplace. We have learned many things by running free software these past few years.

Flexibility

The claims of the free software folk about "free" referring to freedom rather than price is really true. For us this has been the key to the print system. Because we had the source, we were able to modify the code to better meet our needs. To the free software people this sounds obvious but in the corporate world where fully supported is considered the norm, this idea was considered radical. In our case the benefits were immense. The one thing that we learned was in a rather large environment like Cisco the advantages of scale work differently than in a small environment. It probably isn't worthwhile for a company to customize an application exactly to their needs. However in a large corporation such as Cisco, the benefits of customization are great enough that it is economically feasible to modify an application to better suit the needs of the company.

The Final Authority

The downside to this flexibility was that because we were running custom applications, we were the final point of escalation. It is a humbling and sometimes scary thought that if a problem comes up you are the one that has to solve it. You are the final authority on the matter. It requires the corporation to have more faith in its own people. By running custom software, the company is in effect saying that their technical people are competent enough to handle anything that comes up. In many ways, this situation is not that different than that of a commercial piece of software. Somewhere within a commercial software company's organization, there is some engineer who is ultimately responsible for resolving problems with a piece of code. Code is not fixed or modified by corporations. Code is modified by people. The only difference is who that person works for.

The situation has only come up a couple of times. It is scary knowing that no matter what you must find the bug and fix it. I remember having to find a file descriptor leak in LPD. It took me a couple days to actually find it and I felt the weight of the world those two days. It was very sobering. On the other hand one of my coworkers installed a piece of software from the OS vendor. On the dev machine it

installed cleanly with no questions asked. However on the production box, after the files were put into place the installation asked to reboot the machine. There the sysadmin was, sitting in front of the package installation software with a little X-Windows message box saying, "Hit OK to reboot the machine." This just happened to be during quarter end processing and he certainly didn't want to reboot the machine right then. Without the source to the application or the installation script within the package, he didn't know if he could exit the software installer without rebooting the machine. He didn't even know for sure if the machine was in a stable state to keep on running. To say the least he was rather concerned. He called the vendor and their support people, who had never looked at the application source didn't know what would happen either. When they tried to install the package on their test machine it didn't ask to reboot the machine. It took them several hours but they were finally able to replicate the situation and they did find out that if he closed the application, his machine would have rebooted. It took them even longer to figure out that the machine was still stable and how to get out of the package installer without having the server reboot. My co-worker was on tenterhooks during this whole time, and there was nothing he could do to fix the situation. He has to sit there and wait by the phone for the vendor's support people to call him back. I realized then and there, that I couldn't have handled that situation as well as he did. It would have been hard for me to allow the fate of my responsibility to be determined by someone that I didn't know and didn't necessarily trust. At least when I had to find the bug I trusted my own skill. I was afraid but I was in control.

Supportability

The biggest fear of the corporation is that all the developers who understand the application and who have worked on it leave the company and they are left with a completely unsupported unsupportable application. The way that we found to address these fears is to take all of our work and release it to the Internet under the GPL. This has many benefits not the least of which is it benefits the rest of the Internet community.

In having worked with both commercial and free software, I must say that there is a difference. Most of the commercial software is distributed in binary format rather than source form and this difference strikes at the heart of one of the problems associated with custom applications developed in house. I call this problem the "precious binary" problem. With many commercial pieces of software, they are built in carefully controlled environments. The developers do whatever it takes to get the software to compile and run properly and then they have the one precious binary which they package up and distribute. However, it would take a great deal of time and effort to create that binary in another environment. This is one of the problems with software developed in house. Most of the time, the developers create the binary on

their machine and then they distribute it out to their little portion of the world. A potential problem for a company could arise if the developer who created this precious binary leaves and no one can get his source to compile. By distributing it out to the Internet in source format, we the developers have to make it easy for someone else to create a binary. Since the compilation process is done many times with people who have different environments, the compilation process is well wrung out so that it can be repeated by people other than the principle developers.

Another problem that plagues programs developed in house is that source is all too easily lost. A developer leaves and the source trees are left stagnant for a while and eventually they get lost or deleted. If the source is handed out widely then the chances of it getting lost are much smaller.

Another problem for custom applications is that they all too frequently are not documented very well. The person who wrote it knows how to use it and never quite finds the time to write down the documentation. When you release something to the Internet, other people are going to begin to use it. They need the instructions and so the developer must take the time to write them down so that others can make use of their work. Also if the developer hopes to reap some of the rewards of Open Source development and have someone else contribute bug fixes to his program, then he will need to document his source to some degree or another. Also since you know that your peers will be looking at it, you try harder to make sure that the code is good and clean.

Another benefit of releasing code developed for in house applications is that other people will test it more thoroughly than you can. For example, one of the programs that I released did SNMP queries on printers. I thought that I had tested it very extensively but there was one set of SNMP that crashed one kind of printer.⁴ I was one day away from releasing code that made heavy use of that function to all of Cisco when someone reported that problem to me. It saved me from a significant print services interruption.

A common argument against using custom software in a corporation is that if the developer leaves then the company has no hope of finding someone else who has used it. With off commercial software they can hire someone who has used that software before. If the software is released to the Internet, the company can potentially hire someone that has used the program before.

Another way that we convinced management that releasing software to Internet was in its best

⁴The command `npadmin --languages` crashes HP 5M and 5N printers with a 79. This turns out to be a bona fide bug in HP's 5M and 5N printers rather than a problem with `npadmin`. At the time of the writing of this paper HP has yet to provide a fix for this problem.

interest was that once the software is released it somehow becomes personal to the developer. I know that I will be working on the software I wrote for Cisco long after I leave Cisco. Even though it is technically owned by Cisco, it is still my baby. I would never do this for a piece of software that remains locked within a company.

Finally there is the PR value, Cisco likes to hedge its bets. In the early days, that meant that they supported Appletalk, TCP/IP, and IPX/SPX. Now it means that they don't care which wins, DSL or cable modems, they make equipment for both. While they have a strategic relationship with Microsoft, they are also "Empowering the Internet Generation" and so they want to be seen as a friend of Free Software. If Linux ever supplants NT they will be able to point a finger back at the print project and say that they have been a long time friend of Free Software.

As a lowly developer I do not know how much stock they put in any of these reasons. I do not know what was in the managers' minds when they accepted our proposal. However, all of the above elements were in our sales pitch to management.

Author Information

Ben Woodard is 27 years old and lives by the beach in Santa Cruz, CA with his wife Nina and his cat Folger (he's black and wakes you up in the morning). He a Linux fan who loves the wide open spaces provided by OpenSource software and he works as a "System Administrator" at Cisco Systems Inc. but does everything he possibly can including recoding applications to avoid doing any real system administration.

Conclusion

Managing a large organization's printers can be done with a rather small staff. It just requires quite a lot of ingenuity, the flexibility of Unix, and some sysadmins that really are truly lazy through and through and will do anything to get out of boring work.

Reach the author at <bwoodard@cisco.com>.

Large Scale Print Spool Service

Ignacio Reguero, David Foster, and Ivan Deloosse – CERN

ABSTRACT

The paper describes a project to enhance the print service for CERN.

The printer infrastructure consists of over 1000 printers serving more than 5000 Unix users running on workstations of various brands as well as PCs running Linux. In addition, the infrastructure must serve more than 3000 PCs running Windows/95 and NT 4.

We support a large number of printer manufacturers, including HP, QMS, Tektronix, Xerox and Apple.

Lightweight print clients are provided for all the supported platforms and transparently distributed using the ASIS software repository and the NICE application architecture. They may be used as "drop-in" replacements of the standard vendor clients. Compatibility with older CERN lightweight print clients is provided. Printing with standard vendor clients is also possible.

Administrative tools are provided for the general management of print servers and in particular for replicating server configurations and monitoring spool file systems.

The service offers a high level of scalability and fault tolerance, since it has no single point of failure in the server back-end.

Background

CERN [3] is the Laboratory for Particle Physics Research on the Franco Swiss border near Geneva. It is funded by nineteen European countries with a strong participation from other countries like Israel, Japan, the Russian Federation, Canada, and the USA. Some 8000 physicist and engineers work on the CERN site at any one time.

The Printing Problem

The previous printing solution for CERN was based on a single, now obsolete print server (a DEC-station running Ultrix). The print server was acting as a print spool server and Appletalk protocol converter for the whole site. This situation was leading to unacceptably poor availability and increasing printing delays. Furthermore the configuration was very complex as it used two different Appletalk stacks. One using CAP and a Shiva box to deliver IPTalk and another one with Netatalk to deliver EtherTalk. CAP was required to support the older Apple LaserWriter models like the Personal LaserWriter NT.

Whereas the printer server served mainly Unix clients, printing services for Windows PCs had to go mostly through Novell servers and sometimes through Windows NT servers thus making debugging and queue status monitoring very difficult. Some printers were being served simultaneously by a number of print servers and in some cases print jobs were passed between servers. This resulted into an extremely complex situation with many opportunities for problems.

Centralized management, Heterogeneity and Size of the Installation

CERN offers centralized computing services to a very varied community. This explains the fact that little control was possible on the model of printers which was bought, therefore a large number of different printer models, including those of HP, QMS, Tektronix, Xerox and Apple printers, is in use. These printers are accessed from a variety of Unix workstations such as Sun, HP, DEC, IBM and SGI, as well as PCs running Linux, in addition to the PCs running Windows/95 and NT 4.

For historical reasons CERN has a very large population of Appletalk based printers. These still constitute around 60% of the current total population of around 1200 printers. The previous printing service was needed to provide Appletalk conversion for printing from Unix hosts to Appletalk-based printers. The service was then expanded to support a lightweight print client that would eliminate the need for a spooling configuration on the Unix clients. The main problems with this service were the single point of failure, the lack of scalability of the server, and the complexity of managing the PC printing environment with Novell print servers, which provided parallel print services with interconnections.

At the end of 1997 a task force was set to define a new printing architecture for CERN that would be more manageable by simplifying printing operations, removing single points of failure, offering better scalability, and easing integration of new printing technologies.

Commercial Software and Commodity Computing

The more important requirements of the printing architecture are listed below:

- Reduction of the number of print servers servicing a particular printer to (ideally) one.
- Simplification of queue status monitoring.
- Reduction of the number of Netware print servers to one to service IPX-only printers, and the eventual deprecation IPX-based printing.
- Good Appletalk printer support.
- Support for both Windows and Unix clients.
- Preferably use a single protocol to interface to the server queueing back-end.
- Flexible architecture, allowing for easy inclusion of new technology, such as the Internet Printing Protocol.
- Automatic handling of all printer driver issues on the Windows clients.
- Lightweight clients, i.e., without spooling, to simplify client configuration.
- Use of mainstream software and commodity hardware.
- Easy system management.
- Elimination of single points of failure.
- Scalability for thousands of queues with a limited number of servers.

In summary, the aim is to reduce the complexity and diversity of the printing solutions currently being used while being able to support the diverse range of client systems.

Several products were reviewed, but were rejected due to their inability to support the large number of different printer types in use at CERN, or due to their inadequate client system support.

Most of the software solutions relied on Windows NT Print Servers. The scalability of these servers as specified in [17] is of the order of 40 queues per server. However, we have reports of Windows NT servers serving on the order of hundreds of printers. In any case this is not enough for our requirements. Also the support for Appletalk and IPX printers is of unknown quality and the NT printer server will not serve Windows 3.1 users. The inability to use filters is another serious limitation of Windows NT servers.

Finally, the task force arrived to a consensus based on the following conclusions:

- Enterprise printing is still dominated by custom solutions based on Unix servers.
- We cannot buy a complete packaged solution suited to our needs.
- The solution should run on a leading Unix implementation on commodity PC hardware.

The solution finally implemented relies heavily on Unix Public Domain products under GPL [10]. The high quality of products like the LPRng spooler [14, 13] has allowed us to build a very reliable system that is flexible enough to meet our scalability requirements and to support our heterogeneous environment.

Print Server Environment

Choice of Printing Protocol

RFC1179 [11] was chosen as the protocol for the print clients to interface to the server back-end. This is the protocol of the BSD lpd printer server. There are several reasons for this choice:

- RFC1179 is the only non-proprietary standard available today.
- The scalability of connection-oriented protocols like SMB is limited to less than hundred queues per server.
- The management of connection oriented protocols can be complex. In the past we often saw that Windows PCs on some network segments were losing their connections to Novell print servers, thus requiring a manual user intervention and reconnecting to his printer.
- It is available and easy to implement on all our supported platforms.

Appletalk Support

Although we have tried to migrate from Appletalk [1] to TCP whenever possible, the server back-end has to interface to both TCP and Appletalk printers. As more than 60 % of the printers still are on Appletalk this is an important requirement.

There are two main Appletalk protocol types on Ethernet: IP Talk, which is the transmission of Appletalk DDP packets inside IP UDP packets, and EtherTalk which are Appletalk protocols transmitted in specific type Ethernet packets.

Our goal was to support our whole Appletalk printer population with a single Appletalk stack running native EtherTalk phase 2. For efficiency reasons, we required a kernel-level implementation of Appletalk, such as in the Linux kernel DDP module.

We tried to avoid the older IP Talk protocol because it requires maintaining a static routing configuration and special software on a gateway, like a Shiva FastPath to translate IP Talk packets to/from EtherTalk or LocalTalk.

After we first installed the Netatalk package [15], we found that the PAP (Printer Access Protocol) client would not support the idiosyncratic behavior of old Apple LaserWriter printer models, like the Personal LaserWriter NT, hence we had a look at the Columbia Appletalk Package CAP [4].

CAP V6.0 with patch 198 supports EtherTalk using the Linux kernel DDP module and the papif CAP PAP driver successfully runs all our population of Apple LaserWriters including the old models.

However, in both cases, the amount of multicasts required to run Appletalk would break the driver of our Ethernet card. This was the Intel EtherExpress Pro 100B. More specifically, the eepro100 driver generated timeouts and reseted itself before processing the multicast lists required for our Appletalk network, so

the driver was modified to disable hardware multicast filtering.

In the current version of the driver the author has added a module-settable variable "multicast_filter_limit." It may be set to '0' to disable the multicast filter and set the chip to Rx-all-multicast mode, whenever any multicast address is accepted.

Spooling Software

LPRng is our choice for print spooler software to support RFC1179 printing. It is mostly compatible with the lpd BSD print spooler while having a large number of enhancements and new features, some of which are exploited in our project. Features most relevant to us include:

- Lightweight print clients with both SysV and BSD flavors, requiring no databases or spooling setup in the clients.
- Better security, since no SUID root is required in the client software.
- Improved permission and authorization mechanisms: a set of rules determines the type of actions that a given user can perform.
- Improved filter model with support for parameters, and for separated banner page generation, and accounting filters.
- Support for dynamic redirection of queues.
- Excellent debugging facilities.
- Flexible customization model.

Filters

When printing using the BSD lpd model, the server passes the files through a filter that processes them according to a format specified in the control file and the type of device. The processing includes formatting, accounting and banner page generation.

LPRng supports a superset of the BSD printing filter model and allows greater flexibility. In particular, one can specify parameters for filters in the printcap database, and separate functions such as accounting, banner generation, etc. into different filtering programs. This feature was exploited mainly for the banner generation programs.

It should be noted that print jobs coming from Windows clients that use the "NICE Printer Wizard" go through the filters in literal mode, i.e., no formatting is required by the filters since it has already been done by the Windows driver. The filters still have to

do other tasks, such as handling communications with the printer, error reporting, accounting, etc.

apsfilter

We have chosen LPRng's *apsfilter* as the base for our default input filter as it allows us to cleanly integrate different printer models within the spooling system.

The *apsfilter* is a modular script written in Perl [16], which allows us to have a generic behavior. It supports various communication protocols by using different drivers as back-end of the filter.

A *pstext()* function was implemented in *apsfilter* that uses other formatting programs, such as *a2ps*, *psf*, *ascii2ps* and *enscript* to emulate the formatting behavior of the former printing system. This was required for backwards compatibility. *pstext()* supports legacy formatting mechanisms, such as form codes and Fortran Control Characters.

Additional work was done to handle back-end filter errors by getting the printer status from the SNMP agent in the printer log and sending it by mail to the user that originated the print job, thus helping to debug the most common printer operational problems.

papif

We currently use the *papif* PAP driver of CAP as the back-end for Appletalk printers.

This filter was modified, to make the parameter list compatible with LPRng, and to improve status and error reporting.

CTI-ifhp

We make use of the *CTI-ifhp* filter distributed with LPRng to support not only HP-compatible printers, but most TCP printers.

This filter was slightly modified to use more portable tray selection PostScript commands. i.e., something like Listing 1 rather than Listing 2.

This modification was coordinated with logic added to the *apsfilter* so that some parameters are passed to the *CTI-ifhp* filter in order to change the default tray in the HP LaserJet 5Si Mx so that the default tray is the A4 tray, when having A3 and A4 paper sizes.

```
"<</DeferredMediaSelection true /MediaPosition 0>> setpagedevice\r\n"
```

Listing 1: More portable tray selection.

```
"statusdict begin 0 setpapertray end\r\n"
```

Listing 2: Less portable tray selection.

```
"<< /Policies << /Pagesize 2 >> >> setpagedevice\r\n"
```

Listing 3: Avoiding tray switching when paper is unavailable.

The logic in question also can select the printer tray according to the name used for the printer name as described in the *Multiple Tray Handling* section.

In order to avoid tray switching when no paper of the right size is available, we had to change the `PageSize` entry of the `Policies` dictionary by means of a command similar to that of Listing 3.

Unfortunately this is not always successful, since this sort of dictionary can always be overridden by further additions to the dictionary within the job. This is because the PPD files for many HP printers add a `<</DeferredMediaSelection true>>` dictionary entry that disables the `PageSize` policy.

The behavior of this filter may be customized by means of command line options in a very flexible way. For instance, some models such as the HP DeskJet 1600CM, do not support certain PJI commands. The CTI-*ifhp* filter may be configured to support them by means of the options in Listing 4.

These option settings disable the use of the PJI commands for status and page counts and use a PostScript command instead.

The minimal common denominator case may be supported by the `-Tstatus=off` option that treats the printer as write only device. This option may be also useful to support printers that are not bidirectional, when not getting page counts is not an issue.

phaserif

The *phaserif* filter by Al Marquardt et al. is used as back-end for Tektronix Phaser 450 and Phaser 300X models.

pfilter

We are currently testing the *pfilter* by Steve Wilson for Tektronix Phaser 560 support.

Multiple Tray Handling

Multiple tray printers are supported by means of `printcap` entries, an example is given in Figure 1.

The first entry resends the print jobs to the real device queue with a tag describing the characteristics

that are required. In this example, the tag is `aps-PS_600-A3-auto-mono`. This informs to the *apsfilter* which parameters should be passed to the back-end filter in order to select the correct tray. The second `printcap` entry describes the real device queue. All the jobs have to go through this same queue independently of page size or requested tray.

Figure 2 is an example of logic used to analyze the tag in order to determine the tray and pass it to the back-end filter.

```
if( $commprog eq
"/opt/lprng/lib/filters/ifhp" ){
  # choose a paper tray
  if ($opt_Q =~ /.A3/i ){
    $opt_Z = "lower,fixtray";
    $papersize = "A3";
  }
  elsif ( $papersize =~ /A4/i ){
    $opt_Z = "upper,fixtray";
  }
  elsif ( $papersize =~ /A3/i ){
    $opt_Z = "lower,fixtray";
  }
  else { $opt_Z = ""; }
}
```

Figure 2: *apsfilter* logic for multiple tray printers.

Banner Generation

The `-Tbanner=off` option has been specified in the output filter so that banner generation can be handled by an independent *bp* filter.

A `b1` entry has been set in `lpd.conf` as follows

```
b1=$-H:$-n Job: $-J Date: $-t
```

This entry determines the parameters passed to the banner programs.

Banner programs were generated to print the CERN logo, and user and job information in a way compatible with the former printing system. They

```
31-s-012-a3|31_s_012_a3|31-S-012-A3|31_S_012_A3:\
qq:lp=31-s-012@print1:force_queue=aps-PS_600-A3-auto-mono

31-s-012|31_s_012|31-S-012|31_S_012:\
qq:server:direct_read:mx#0:ps=status:pw#2:lp=137.138.34.47%9100:\
sd=/var/spool/lpd/31-s-012:\
if=/opt/lprng/lib/filters/apsfilter/apsfilter:\
of=/opt/lprng/lib/filters/ifhp -Tbanner=off:\
bp=/opt/lprng/bp/psbnrhlj5simx:\
vf=/opt/lprng/lib/filters/ifhp -c:
```

Figure 1: The `printcap` configuration file entries for multiple tray printers.

```
-Tsync=off,infostatus=off,pagecount=off,forcepagecount=on.
```

Listing 4: CTI-*ifhp* filter options for HP DeskJet 1600CM.

contain PostScript code to switch to the “yellow paper” tray, if it exists, so that the banner is printed on paper of a different color.

For Appletalk printers, the banner program writes out the banner into .banner in the spool directory for the papif input filter back-end, which can then later print it.

Accounting

Although no direct charging is done, accounting is enabled and a Perl script was written to process the accounting log files. It generates daily reports detailing usage, such as pages per user per printer, and pages per host per printer. It also generates summary reports of integrated usage for the last month.

Per printer information is accessible through a Web interface described in the *Web Interface* section.

Server Clustering

Clustering is required in the server back-end in order to guarantee scalability and resilience to failures.

Several alternative architectures were considered, for instance the one in which a “master” front-end machine distributes print jobs to an array of workers. This solution was rejected because it would still

present a single point of failure and it would complicate the management of print queues.

As a consequence a simpler architecture had to be developed with the following characteristics:

- An array of print servers which are configured with similar queue definition files (printcap, etc).
- Printer queues are distributed among the servers to balance the workload.
- A printer is served by a single server at any given time.
- An external naming service directs the print clients to the server assigned to each specific printer.

The advantages of this design are numerous:

- Queue management is easy as only one server sends jobs to the printer. Contention is not possible.
- In case of failure of one of the servers, a reconfiguration of the naming service database suffices to reallocate the queues served by the failed server onto active ones.

```
printername IN CNAME servername.cern.ch.
```

Listing 5: DNS configuration sample.

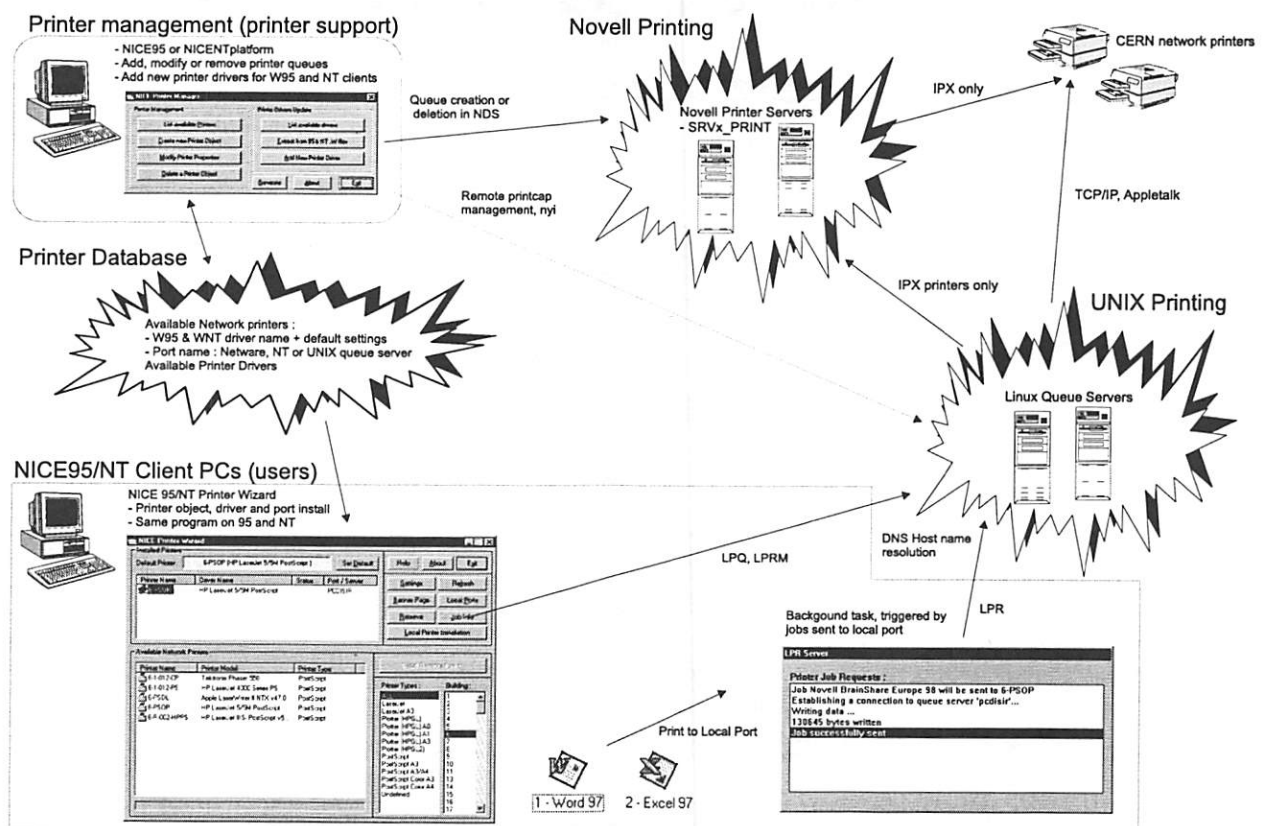


Figure 3: PC printing architecture overview.

Naming Service Setup

The Domain Name Service (DNS) has been chosen as dispatcher since it is easily accessible from the client software. Moreover a high availability DNS server infrastructure already exists at CERN.

A `print.cern.ch` DNS hierarchy was setup, such that for each printer, an alias of the form `printername.print.cern.ch` is available.

These aliases map printers to the server which serves their specific printer queue.

To generate the aliases, a DNS configuration file is required with entries as in Listing 5.

The caching lifetime for the entries was set to 10 minutes. The configuration is reloaded into the server also each 10 minutes. This determines the latency when reconfiguring the system.

A fall-back DNS configuration file is available for each server. In the fall-back file all the printers in the dead server are assigned to the other servers. Hence, when a server is down, the DNS is reconfigured by swapping the configuration file so that the queues can point to the surviving nodes.

Both Unix and Windows print clients that we support address the server through this DNS hierarchy as described in section named *The Unix Print Clients*.

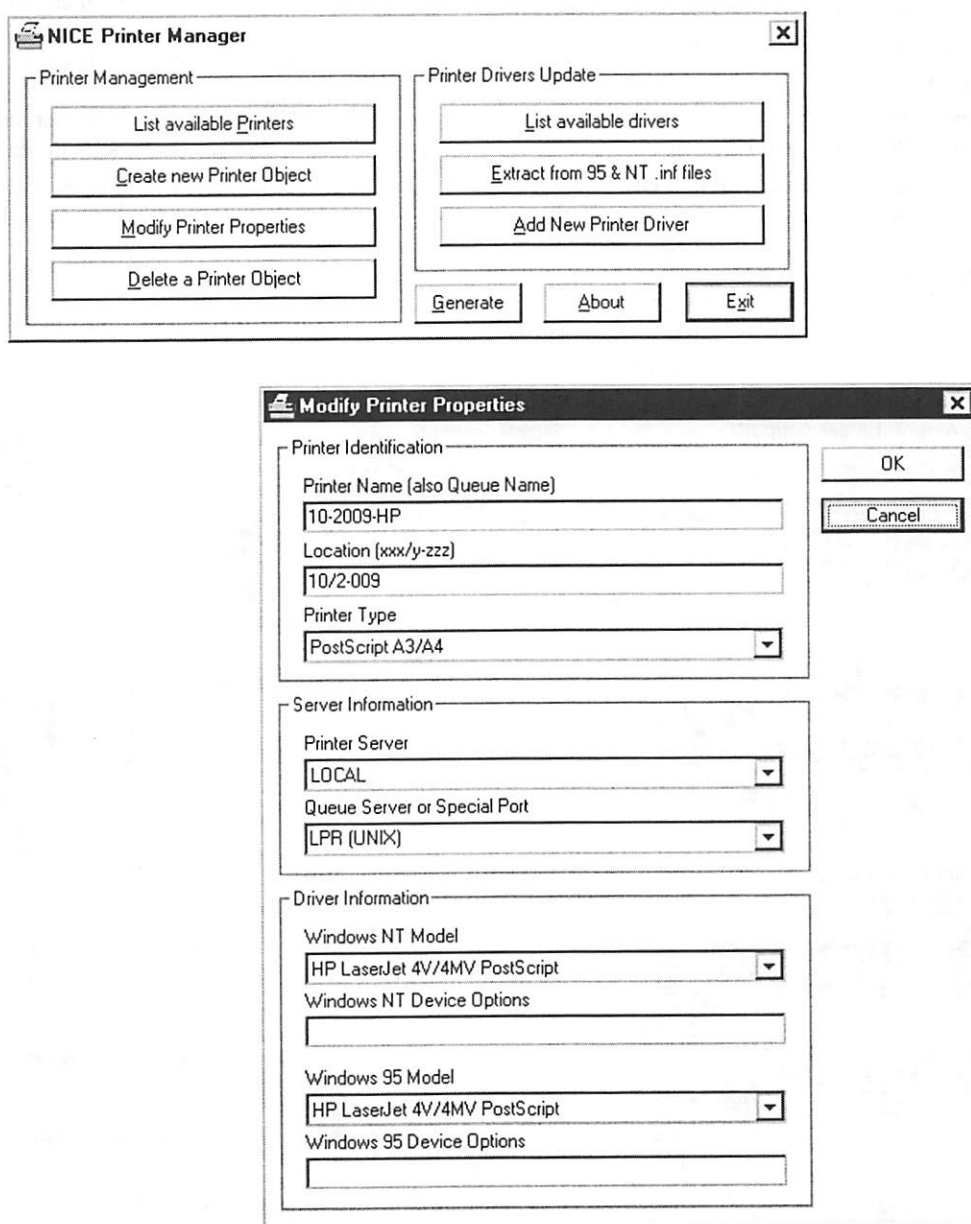


Figure 4: Printer Manager.

Scalability and Fail-over

The architecture implemented can support with two or three PC servers, our current printer population of around 1200 printers, for a client system population of some 2500 Unix workstations, servers and Linux PCs, 3000 Windows PCs and 1700 X-terminals.

The fail-over mechanism currently requires intervention of an operator and results in a slightly degraded mode, but the system is simple, robust and easy to manage.

The PC Printing Architecture

Introduction

From the PC/Windows point of view there is no commercial software available to integrate PC network printing into a Unix printing architecture. In order to avoid the deployment of platform dependent printing architectures (e.g., Netware and NT for the PC world) the CERN standard Unix printing server architecture has been adopted for the Windows clients.

The architecture for Windows 95 and NT clients can be grouped in the following building blocks:

- A common database and driver repository defining all CERN network printers (queue name, W95/WNT driver data,...) which is stored on a central application server.
- Administration tools: a set of programs used by the printer administrators to install, modify or delete network printers/drivers.
- Client software: Windows based programs running on the client PCs enabling the user to connect to any CERN network printer in an easy way.

The "NICE Printer Wizard" is the key application for printer installation and configuration on the PC platform.

The PC printing architecture is shown at Figure 3.

The printer database and repository

MS Access was chosen as the database medium because of its good integration with the PC world and the powerful development environment. The database contains the following information:

- **Supported printer drivers:** In addition to the standard printer drivers included in the W95 and WNT distribution CDs, the database holds information about third party drivers added by the administrator. For every driver, the database has a reference to the corresponding .inf and driver files. These files are stored in the printer data repository and used by the client software when a user connects to a printer.
- **Available network printers,** defined by:
 - **The name of the printer,** identical to its queue name on the Unix server
 - **The printer driver name,** the name of the .inf file as well as the path to the

driver files for the W95 and WNT platforms.

- **The printer type.** Selection is made from a predefined list in the database.
- **The default printers settings** as defined by the administrator.

Administration tools

The printer administration tool, called the "NICE Printer Manager," has the following features:

- Possibility to add third party printer drivers. Any new printer driver which is not part of the standard W95/WNT distribution CDs can be added in an automated way. All information required to install this driver on a client PC is added to the database and the necessarily files are copied to the repository by the "NICE Printer Manager."
- Management of the network printers. For every CERN network printer, the "NICE Printer Manager" allows the administrator to:
 - enter the printer name.
 - select the W95 and WNT driver name from the supported driver list.
 - choose the printer type.
 - define the default settings.

The data stored in the printer database by the "NICE Printer Manager" is extracted to binary data files after every modification. For performance reasons these files are then used by the client software instead of the database itself.

The Printer Manager is shown at Figure 4.

The Client Software

The client software is split into two components:

- **The "NICE Printer Wizard":** a graphical interface which enables the user to connect to all network printers defined in the database. Connecting to a printer from W95 or WNT implies installing the driver files on the client PC, importing the default settings and creating the printer port. In this architecture, the printer port will point to a file on the local hard disk. The "NICE Printer Wizard" presents the available printers organized by building number and printer type, so the user knows exactly the capabilities and location of the nearest printers. Other options include the configuration of the printer settings and banner page. The "NICE Printer Wizard" is shown at Figure 5. On request, the printer wizard returns information about the current jobs for every connected printer with the possibility to delete them. Note that the job information is queried from the corresponding Unix printer server by implementing the lpq and lprm protocol inside the Printer Wizard. This gives the big advantage that a PC user has an overview of all current jobs on the connected printers, even sent by non PC platforms (e.g., Unix clients).

Another benefit in a heterogeneous client environment gained by having a single server (and hence queue) for a particular printer.

The Job Info window is shown at Figure 6.

- **The LPR Service module:** This task is permanently running in the background on every Windows 95 and NT PC and is the gateway between the local printer ports and the Unix printer server. It is triggered by the Microsoft Printer Change Notification Mechanism when the user prints a job from an application program. As defined by the "NICE Printer Wizard," the printer port sends the data to a local file on the hard disk, containing the name of the printer. As part of the communication convention between the "NICE Printer Wizard" and the "LPR Service," this background task

knows exactly where to find this file and will then transmit the contents to the Unix printer server using the LPR protocol.

The "LPR Service" window during a job transmission is shown at Figure 7.

We use the NICE [2] architecture to deliver and maintain the software so that it appears in the start menu of all our supported platforms by means of a number of replicated application servers to which all users connect.

The Unix Printing Architecture

The Unix Print Clients

The print clients supported are the LPRng clients which have been modified to allow the use of DNS to locate the print server. In particular, the modification

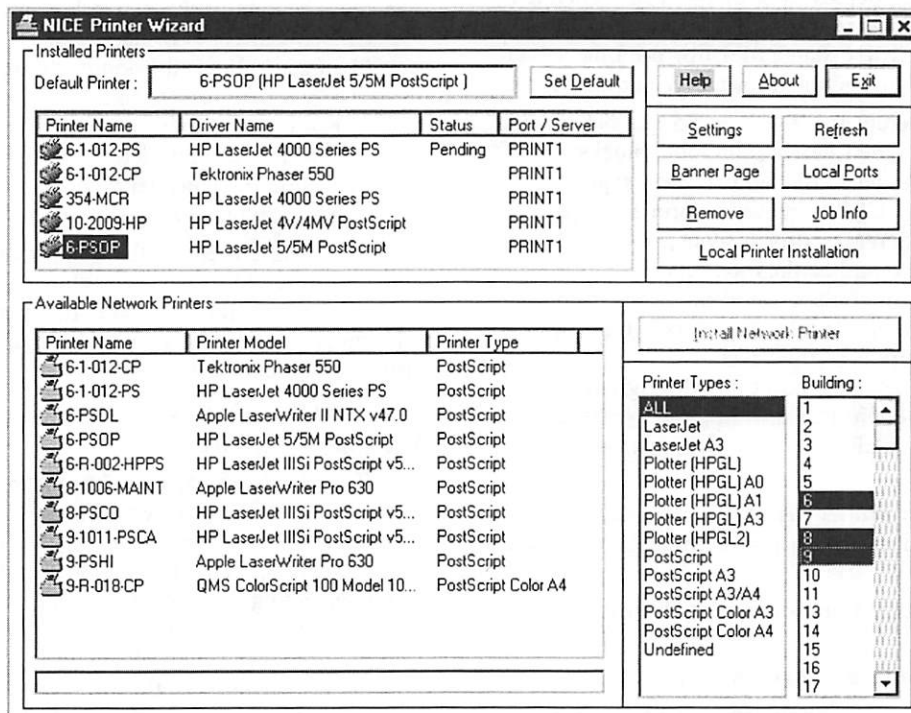


Figure 5: The "NICE Printer Wizard."

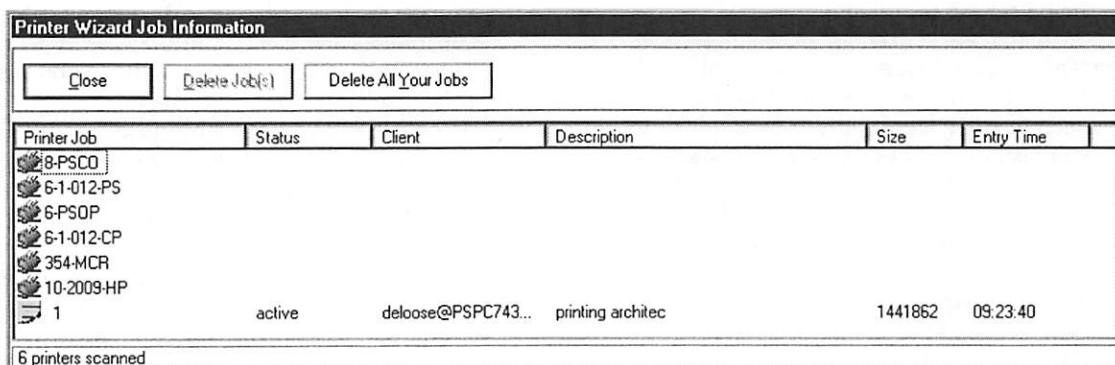


Figure 6: The Job Info window.

ensures that printer names that are specified as -Ppname are translated into names like pname@printername.print.cern.ch, thus making the print server back-end scalable in a way that is transparent to the print clients.

The DNS resolution mechanism is shown at Figure 8.

The LPRng clients have the interesting property of being lightweight. This means that they send the job directly to the print server, not requiring any spooling configuration on the client.

We use the ASIS [7, 8] repository to distribute the software so it appears in /usr/local/bin on

all supported platforms via a distributed file system such as AFS, NFS, or as a controlled local copy.

The supported Unix platforms are Sun Solaris, IBM AIX, HP HP-UX, DEC Unix, Linux on Intel and SGI Irix. We plan to replace the print commands distributed with these systems with the LPRng clients.

Support for Standard Vendor Clients

Although we do not recommend to use vendor specific print clients, LPRng in our servers has been configured with the "fix_bad_job" configuration option, so that it supports the widest spectrum of clients possible.

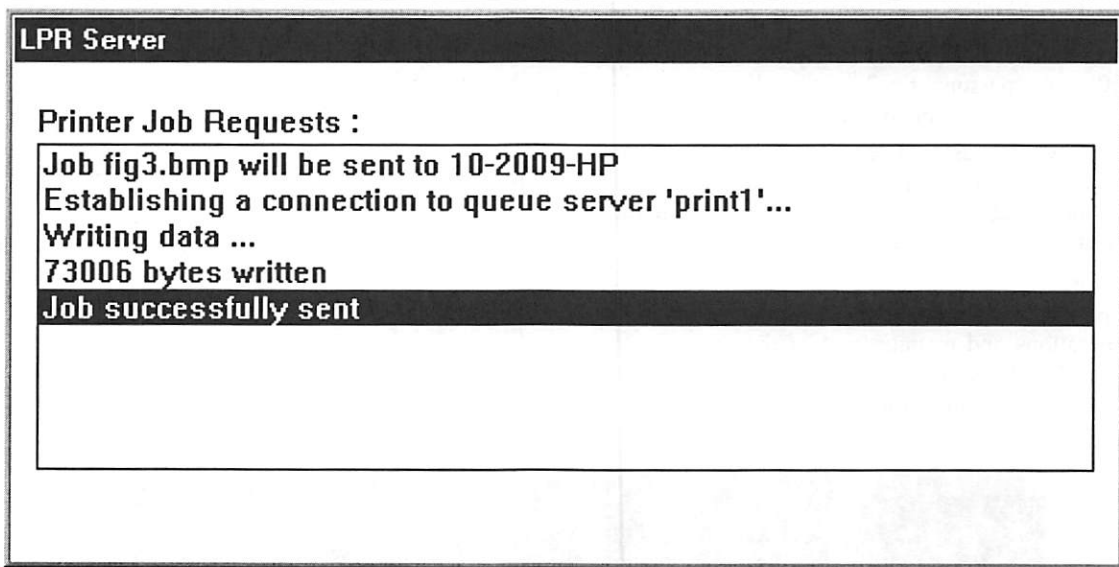


Figure 7: The "LPR Service" window during a job transmission.

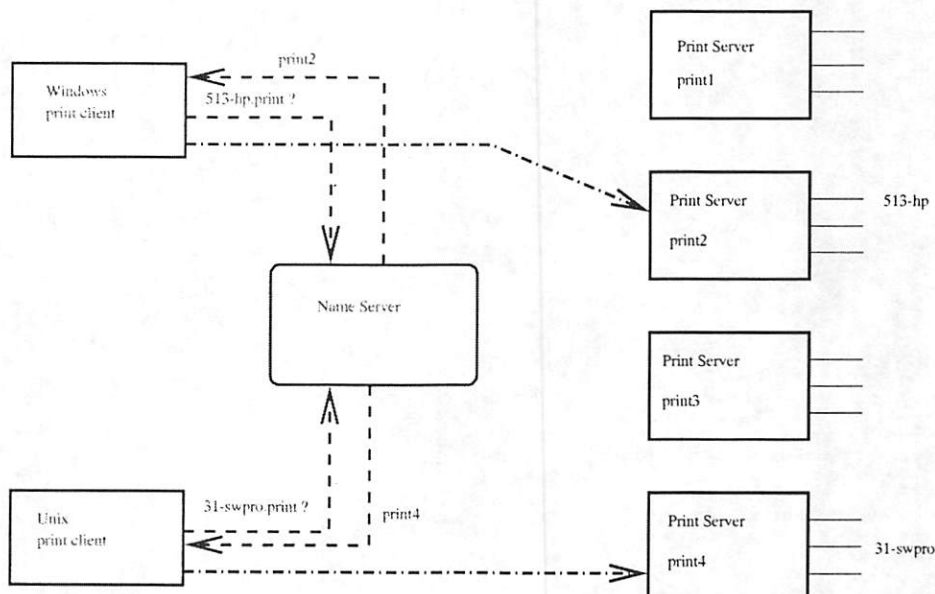


Figure 8: The printer resolution mechanism.

For instance a BSD `lpr` print client could access the print servers by using the following `/etc/printcap` entry:

```
prntername:lp=:\  
rm=prntername.print.cern.ch:\  
rp=prntername:\  
sd=/var/spool/lpd/prntername:mx#0:
```

Backwards Compatibility

The `xprint` lightweight print client has been used at CERN since more than five years. Since it uses its own protocol, it requires its own `xprintd` daemon machinery in the server to format the control file and spool the print jobs.

In order to simplify the printing system, it was decided to stop supporting the `xprint` code. Instead we use the interface in the LPRng print client to support different "personalities" of the client to implement a `xprint` like look and feel.

Some work was also required in the Perl script for `apsfilter` to support specific old formatting options, including forms and Fortran Control Characters as described in the *Filters* section.

Management

One of the main goals of the project was to simplify operations and management of the printing service. Therefore several tools were provided to ease the migration and the management of the service.

Migration Tools

The main migration tool is the `syncpcap` script that combines `printcap` queue configuration and Appletalk addresses from the old service to generate the LPRng server spooling configuration.

The generated configuration includes the `printcap` file, the spool directories, the configuration for `apsfilter`, the Appletalk address file `cap.printers`, and the DNS configuration.

Security

To reduce security risks, the LPRng client executables are installed as `non-setuid`. In the same line, access to the print servers is reduced to the strict minimum, the `ssh` secure shell has been installed and the `sudo` utility is used to delegate privileges to the operations team.

Authorization

The permissions for LPRng, both to print, manage print jobs, and to view their status, are controlled in great detail by the authorization rules of LPRng in `/etc/lpd.perms`.

The current rule set allows users to manage jobs submitted by themselves, but (unless they are members of the Printer Operations Group) they cannot control other jobs.

These rules are also being used to set access control lists in printers with expensive consumables, so that only certain users can print on them.

```
xterm  
22-1-cor 4-2-002-320 594-r013-tek pisa-lw  
22-cb-printer 4-2-11 595-uge pisa-lw1  
2257-r-hp5ps 4-2-next 595-wa70 pisa-lw2  
23-1-022-ps 4-2021-cm 595-wa98 pisa-lw3  
23-1007-tek 4-3009 596-r020 pisa-lw640  
23-kkg 4-3016-dm 596-r021-lw pollux-orsay  
23-r-005 4-3019-f 596-r027-cm prtshop1  
23-rolf 4-3022-f 598-hpc prttest  
24-1-004 4-3024-f 599-mtsm ps4dr27f  
24-1-8-hp4 4-3026-f 599-r000-lw-ps pythie-orsay  
24-1018-tek 4-3029-lw 6-1-012-cp pythie-ps-orsay  
24-e011-dc 4-3030-f 6-1-012-ps ssdd-161-1  
24-tisdi 4-3cor-f 6-psdl t9-nomad  
24-tisgs 4-3cor-hp 6-psop unige-1  
24-tisrp 4-dgc unige-2  
25-r036-plt 4-doran unige-3  
25-r038-cm 4-dtp1 unige-4  
25-r038-cm-t 4-fi-ppe unige-5  
252-1-hp4ps 4-ps205 unige-6  
252-1-lw 4-qap-laser unige-7  
252-1000-hp3ps 4-r-057-ess van6ps1  
252-atrf 4-r005-ess van6ps2  
26-1-003 4-r046-ess van8ps1  
26-2001-cm 4-r05-cm van8ps2  
26-2001-cmt 4-r13-capro wisc1  
26-2001-hp 4-thcolor wisc2  
[sunadm] project/printing/lwc > 31-1029-dci  
31-1029-dci>lpg  
Printer: 31-1029-dci@print1 '31-1029'  
Queue: no printable jobs in queue  
Status: server finished at 20:58:50  
Filter_status: ifhp Initial page count 235378, final 235382, Total pages = 4, elapsed time 64 secs at Sep 20 20:58:50  
31-1029-dci>status  
Printer Printing Spooling Jobs Server Slave Redirect Status/Debug  
31-1029-dci@print1 enabled enabled 0 none none  
31-1029-dci>
```

Figure 9: Printer status with `lpc` extensions.

Queue Status Information

Operations are greatly simplified already by the fact that a single queue is used for all users of each printer, whether they submit from Windows or Unix. This fact also circumvents the large family of problems that may arise when several servers attempt to connect to a same printer simultaneously.

The standard tools provided with the print clients, such as `lpq`, `lprm` and `lpc`, can be used to manage the print queues.

Any of these commands can be used in any client independently of the location because they have been modified, in the same way as the print clients, to support our DNS print hierarchy.

The permissions are controlled by the authorization rules of LPRng as described in section named *Authorization*.

For compatibility with an older tool, some extensions to `lpc` have been implemented. For instance, a directory in the AFS distributed file system holds all the printer names available as links to the `lpc` program, so that running one of them directly calls `lpc` for that printer. The usage of this interface is shown at Figure 9.

In addition, a WWW interface is offered for these commands. This interface is described in the *Web Interface* section.

Printer Status Reporting with SNMP

A utility script has been developed using the CMU `snmpget` and `snmpwalk` distributed with Linux to retrieve the current status of a printer. This is particularly useful for problem determination.

The status is also made available through the WWW interface described in the *Web Interface* section.

Server Configuration

In order to ensure that the system configuration is exactly the same in all print servers, the installation of these systems has been completely automated.

The `kickstart` utility that is distributed with RedHat Linux is used to automate the basic system installation. This is then linked with the installation of SUE [6, 5]. SUE is a Unix automated system management framework done at CERN. SUE features for LPRng, CAP, and the configuration management scripts install and guarantee the consistency of these elements.

The configuration management scripts insure the correctness of the queuing configuration of the servers. Changes to the configuration must be securely propagated to all servers.

The queuing configuration includes the `print-cap` file, the spool directories, the `apsfilter` configuration and the Appletalk `cap.printers` address file.

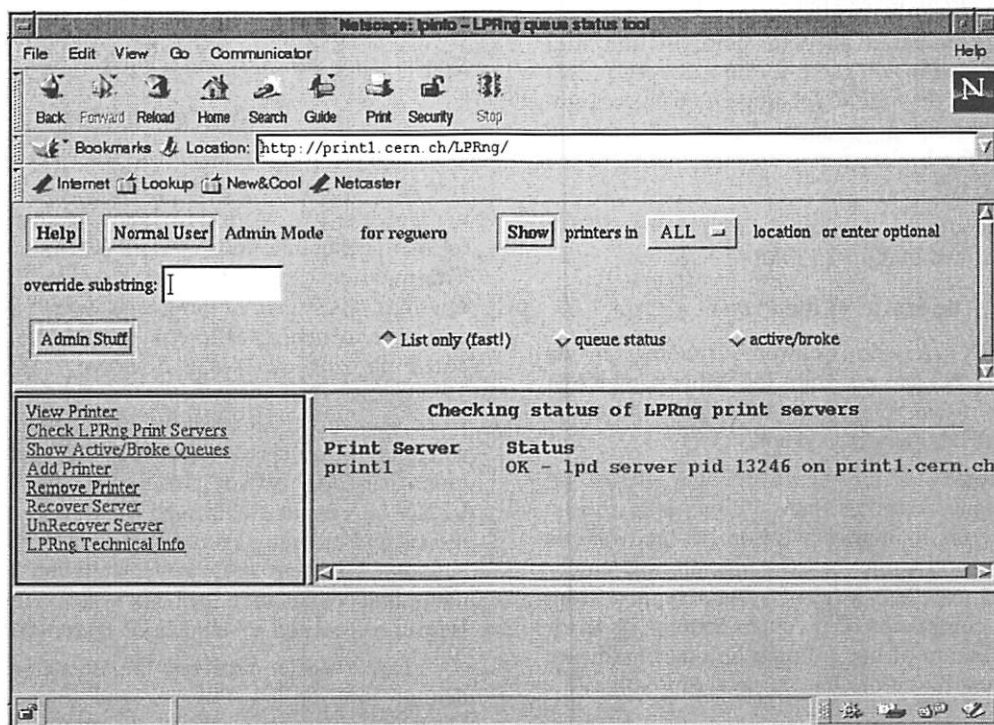


Figure 10: WWW administrative interface with `lpinfo`.

Server Monitoring

All the print servers are included in the CERN specific alarm system. In this system both a heartbeat signal and several other checks on processes and system resources are done. In particular spool space usage is periodically checked this way.

Web Interface

Users are provided a WWW interface to view the status of the printers. The same interface allows the Printer Operations team to actually control the printers. This functionality is based on the `lpinfo` CGI scripts [12].

The `lpinfo` Web server(s) and CGI scripts are run in the print servers.

The following features have been added to this interface:

- View queue status
- View server status
- View SNMP printer status
- View accounting for a printer
- Add printer
- Remove printer
- Move server out of the cluster
- Move server into the cluster

A view of this interface is shown at Figure 10.

Recovery Procedure

For convenience, the DNS server loads the configuration file from a directory in the AFS distributed file system.

Currently, the configuration file describing the print DNS hierarchy is read once every 10 minutes.

We have prepared alternate configuration files for failover situations. Therefore the recovery procedure consists of copying the specific alternate configuration file to the one in production, thus redistributing the queues in the broken server to other nodes of the print server array.

This operation is supported through the WWW interface described in the *Web Interface* section.

The Status of the Project

The PRINT service is currently running on two Digital Prioris PC servers with 256 Mbytes of RAM and 9 Gbytes of disk running the LPRng spooler under the Linux RedHat operating system.

The choice of PC hardware was based on price/performance considerations. Linux was chosen as a leading Unix implementation on PC hardware as it offers very good kernel-based Appletalk implementation, and an excellent level of system support from the network community. However, nothing prevents the implementation of our solution on other hardware or on a different flavor of Unix, such as Sun Solaris.

Deployment Plan and status

The new service is progressively replacing the old printing system.

Most of the PC users have been migrated to the new system, while the Unix clients are currently under test. The migration for the Unix clients will proceed during fall 1998.

Availability

CAP + LPRng + Linux

All these components are publicly available. Their integration is described in this article. If interested in specific details please contact Ignacio Reguero (i.reguero@cern.ch).

Printer Wizard

For information on the "NICE Printer Wizard," please contact Ivan Deloose (ivan.deloose@cern.ch).

NICE

Information on the NICE architecture is available through <http://nicewww.cern.ch> or from David Foster (david.foster@cern.ch).

Future Developments

We are studying how to rationalize the dissemination of status information for Windows clients to reduce the number of `lpq` queries required to show the print queues in the printer wizard.

We would like to implement Web access to server back-end status, as well as user notification using Zephyr [9] messages.

We are considering the possibility of automatically triggering the recovery procedure from the alarm system when a print server is missing.

Author Information

Ignacio Reguero received the "Ingeniero Superior de Telecomunicacion" degree from the Polytechnic University of Madrid in 1985. He studied in parallel two curriculum areas: "Data Communication/Data Transmission" and "Computer Science/Data Transmission." His graduation thesis was "Specification and Implementation of Interpersonal Messaging Protocol, Following the X.420 Standard." He worked in 1986 and 1987 as system engineer for the IBM IN network. He worked in 1988 for "Banco De Santander" as communications systems engineer of a network of more than 2000 offices. Since 1989, he is working at CERN on various fields such as large systems performance and communications, network backup systems, automated system software installation and maintenance and consultancy for Unix system administrators. He can be reached at [<Ignacio.Reguero@cern.ch>](mailto:Ignacio.Reguero@cern.ch).

David Foster received the Bachelor of Science degree in Applied Physics and Electronics from Durham university in 1978. This was followed by the Doctor of Philosophy degree in 1982 for work in real-time multiprocessor control systems for atmospheric

physics experiments. Since 1981, he has worked at CERN on a variety of projects including compiler design for cross platform development, communications to IBM mainframes and management of enterprise PC systems. He has acted as an industry consultant in these fields and his work has been widely published. Currently he is the manager of the PC desktop support team at CERN and is completing a Master of Business Administration at Durham University Business School specializing in large scale IT infrastructure support for knowledge management. He can be reached at David.Foster@cern.ch>.

Ivan Deloose received the "Industrieel Ingenieur Elektronica, optie Telecommunicatie" degree from the Industrial High School of Kortrijk in 1987. His graduation thesis was based on the design of a Digital Video Effects Mixer. He worked in 1989 as development engineer for BARCO INDUSTRIES. In parallel, he developed the electronics for a large video projection system (video wall). Since the end of 1989, he worked at CERN in the sector of operations and controls of the particle accelerators. He was especially involved in a wide variety of software projects in the domain of accelerator controls. In 1995, he became the divisional responsible for office automation and network infrastructure. He is managing the design and architecture of PC based control system software inside the division and for some attached experiments. He can be reached at <Ivan.Deloose@cern.ch>.

References

- [1] Apple Computer, Inc. *Inside Macintosh: Networking*, <http://developer.apple.com/techpubs/mac/Networking/Networking-2.html>.
- [2] CERN. *NICE Architecture*, <http://nicewww.cern.ch/homepage/Documentation.htm>.
- [3] CERN, European Laboratory for Particle Physics. <http://www.cern.ch/>.
- [4] Columbia University, et al. *Columbia Appletalk Package*, <http://www.cs.mu.OZ.AU/appletalk/cap.html>.
- [5] Lionel Cons. *SUE Users Guide*. CERN, the European Laboratory for Particle Physics, 1996.
- [6] Unix Workstation Support CN Division. *SUE Definition Document*. CERN, the European Laboratory for Particle Physics, 1.00 edition, 1995.
- [7] Ph. Defert, et al. "Automated management of an heterogeneous distributed production environment." In *First Conference on Freely Redistributable Software*, pages 1-8, 1996.
- [8] Ph. Defert, et al. "Managing and distributing application software." In *LISA'96 Proceedings*, pages 213-226, 1996.
- [9] Tony Della Fera, et al. 'Zephyr Notification Service' *Project Athena Technical Plan Section E.4.1*, Massachusetts Institute of Technology, 1987.
- [10] GNU. *GNU General Public License*. Free Software Foundation, Inc., 1991.
- [11] L. McLaughlin III. *RFC 1179*. Network Printing Working Group, 1990.
- [12] Alek Komarnitsky. *lpinfo*, <http://www.komar.org/komar/alek/>.
- [13] Patrick Powell. Managing network printers and print spoolers. In *LISA XI*, Tutorial T15pm. Astart Technologies Inc., 1997.
- [14] Patrick Powell and Julian Mason. "Lprng - an enhanced printer spooler system." In *LISA IX Proceedings*, pages 13-24. Usenix, 1995.
- [15] The University of Michigan. *Netatalk*, <http://www.umich.edu/rsug/netatalk/>.
- [16] Larry Wall, Tom Christiansen, and Randal Schwartz. *Programming perl*. O'Reilly and Associates, Inc., 1996.
- [17] *Microsoft Windows NT Server Resource Guide*, Microsoft Corporation, 1996.

mkpkg: A software packaging tool

Carl Staelin – Hewlett-Packard Laboratories

ABSTRACT

mkpkg is a tool that helps software publishers create installation packages. Given software that is ready for distribution, *mkpkg* helps the publisher develop a description of the software package, including manifests, dependencies, and post-install customizations. *mkpkg* automates many of the painstaking tasks required of the publisher, such as determining the complete package manifest and dependencies of the executables on shared libraries. Using *mkpkg*, a publisher can generate software packages for complex software such as TeX with only a few minutes effort.

mkpkg has been implemented on HP-UX using Tcl/Tk and provides both graphical and command line interfaces. It builds product-level packages for Software Distributor (SD-UX).

Introduction

Most end-users do not build programs from source code, but install software using binary installation packages. *mkpkg* helps software publishers develop those installation packages. Building my first complex installation package by hand took me a week, but with *mkpkg* I can build installation packages with roughly three minutes of effort.

mkpkg addresses a part of the software distribution channel that has been largely ignored. Most software distribution systems have focussed on defining the binary package format and the protocols for installing and de-installing software. Most software installation suites have made it very easy for end-users and system administrators to distribute and install software, but they have not addressed the problems of the software packager who is creating binary installation packages.

The software publishing process is illustrated in Figure 1 and includes several actors and steps: the software developer, the software publisher, the distributor, (sometimes the system administrator), and the end-user. The software developer creates the software. The packager is responsible for configuring, compiling, and packaging the software to create the binary installation package that the distributor delivers to the end-user. In some environments system administrators install and manage the software for end-users.

mkpkg helps software packagers create installation packages. Typically, the packager starts with source code that needs to be compiled and installed on the packager's computer. The packager tries to create an installation package that re-creates the installation on each end-user's computer.

Developing a binary software installation package, that is, creating a package that can be installed easily on a computers and have it work properly, is an important and difficult task. Most vendors have developed tools that can accept the descriptions of a software package and create an installation package, but developing those package descriptions is difficult. Package descriptions typically contain the elements listed in Table 1.

Each software installation tool has its own idiosyncrasies and requirements, but they all share these common elements. Many software installation tools also provide other elements, such as system specifications that define which hardware/OS types or versions may install the software.

mkpkg works in conjunction with software distribution tools, such as Software Distributor, by assembling all the elements required by the tool-specific packaging program, such as Software Distributor's swpackage. *mkpkg* is flexible and its back-end can be modified to create packages for different packaging tools. A Linux/RPM port is planned for the future.

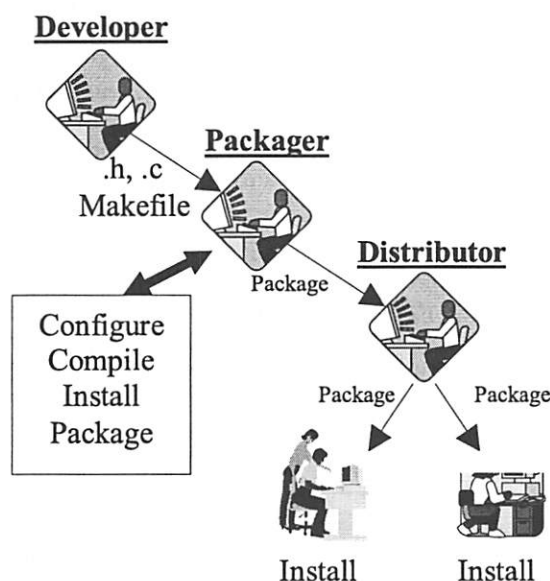


Figure 1: Software publishing process.

Element	Description
title	Software package name
description	A text description of the package and its capabilities
manifest	A list of all the files contained in the package
dependencies	A list of all the other packages required for this package to operate correctly
customization scripts	A set of scripts that are executed on the user's machine during installation or de-installation of the software

Table 1: Elements in packages.

Software Distributor

Software Distributor (SD-UX) is a suite of software installation programs that satisfy the POSIX draft 1387.2 specification. Packagers use *swpackage* to transform a Package Specification File, binary files, installation scripts, and other files into a complete binary package. *swcopy* creates and manages repositories of installation packages, and *swinstall* actually installs software. Software Distributor is the software distribution mechanism for all Hewlett-Packard software for HP-UX and has versions that run on at least WindowsNT and Solaris.

Figure 2 demonstrates the four levels of software grouping in Software Distributor: bundle, product, subproduct, and fileset. A bundle is a collection of products and/or filesets that may be installed as a unit. Bundles were designed to provide customers with one single installation unit for purchased software products, such as the ANSI/C compiler. Bundles may be used to provide a logical grouping by function, such as "web server."

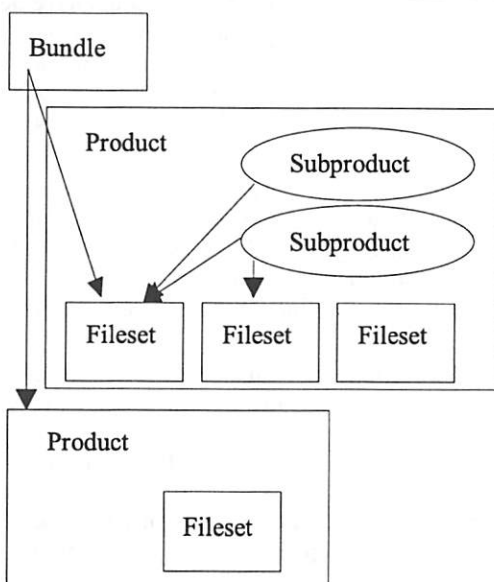


Figure 2: Four levels of software grouping.

The basic unit of software distribution is the product. A product may contain both subproducts and filesets. Subproducts contain filesets and are used to manage logical subsets of a single product. For example, the ANSI-C compiler product might have a subproduct for each language containing all the filesets needed to install the compiler with messages and documentation in the proper language.

Filesets are the atomic units of software distribution and contain a set of files and control scripts. SD-UX installs and configures individual filesets; filesets cannot be partially installed or configured.

Software Distributor has two levels of software distribution: bundle and product. The basic unit of distribution is the product. Software Distributor has several levels of software installation. The basic unit of installation is the fileset, but customers usually install software at the bundle or subproduct level.

mkpkg creates product packages, while *mkbd* creates bundles. Since all filesets and subproducts are created as part of a product, we do not provide a separate tool for creating them.

Installation Tools

There are many methods for distributing binary installations. Each method has various strengths and weaknesses, but most commercial systems provide a similar level of basic operation. The largest UNIX vendors have each developed their own systems for distributing binary software: HP's HP-UX uses Software Distributor, Sun's Solaris uses *pkgadd*, Digital's OSF/1 uses *setld*, SGI's IRIX uses *inst*, and Linux uses RPM. Windows has two standards, InstallShield packages and self-extracting programs.

Each of the commercial systems offers basic services, such as installing and deinstalling packages atomically, tracking and managing inter-package dependencies, and executing scripts during software installation and deinstallation. Most of them also support a variety of more advanced features, such as versions, operating system and hardware dependencies, and interactive installation. For the most part these systems try to make it as easy as possible for system administrators to install software.

Tar

The simplest binary installation package is simply a tar file containing the software. Often such packages include a README file that includes installation instructions. For simple programs and packages, tar files are often sufficient. For more complex packages, much of the burden of correctly installing and configuring the software falls on the end user because the installation process for tar files is completely manual.

Occasionally, software publishers will include installation scripts as part of the tar file and the installation script will automatically install and configure the software for the user. One of the few publishers using this approach is Netscape, who includes an "ns-

install" script as part of the Netscape Communicator tar-file distribution.

RPM

The RedHat Package Manager (RPM) [2,3] was developed for the Linux environment and provides a very nice environment for installing and distributing software. Functionally, it is very similar to Software Distributor; it includes support for inter-package dependencies and control scripts that are executed during software installation.

Users may install software from a local depot or they may install from a remote server over the network. RPM is able to contain binaries for multiple platforms within a single package, and it can automatically install the correct binaries. Using RPM customers may determine which package installed a particular file, and what software is installed on the machine. Users may also uninstall packages.

RPM has some support for the packager, but it is missing some important features. RPM does not help the publisher develop the package manifest.

Software Configuration

Software configuration is one of the dirty little secrets of system administration. Software that is well configured works well in a broad variety of system configurations and causes few problems for system administrators. Poorly configured software can cause system administrators a great deal of aggravation.

Sometimes the difference between well-configured software and poorly configured software is a matter of tiny details, but there are a few general guidelines.

- Never compile paths into binaries.
- Separate executables, configuration files, and data or log files.
- Use human readable ASCII files for configuration information.
- Follow standard conventions for file and path names as much as possible.

System administrators frequently share file systems between systems, so executables and libraries will often be shared by many systems. However, administrators usually want each computer to have its own version of configuration files, but these may be on a read-only file system. In addition, data and log files are usually not shared between systems and usually must be mounted with read-write permissions. Software configurations must anticipate these kinds of configuration issues.

A fairly detailed set of configuration guidelines is published by the HP-UX Porting and Archive Centre [4].

Software Packaging Process

During software packaging, the publisher must prepare all the elements needed by the installation package. For many small packages, this is a very

simple process, but for larger packages it can be quite difficult. *mkpkg* provides five services during the packaging process:

- Creates the manifest, the list of files to be installed
- Determines the dependencies of this package on other packages
- Develops the install/de-install scripts
- Gathers all the components, as listed in the manifest
- Assembles and produces the completed installation package

Create Manifest

The manifest is a list of all the files to be installed by the package. *mkpkg* can automatically determine which files were installed by the package on the publisher's machine. For small packages it is easy to determine which files belong to a given package, but manual techniques often miss files and make mistakes. For larger packages, such as X11R6 or TeX, it is usually difficult to identify all the files installed by the package and it is critical to include all the files that belong to the package.

Determine Dependencies

Many packages require the presence of other software in order to operate correctly. For example, if cvs uses rcs, then cvs depends on rcs. Packages can depend on other packages for many reasons, but executing programs and linking with shared libraries from other packages causes the two most common dependencies. Publishers are usually aware of the dependencies caused by executing programs, but often overlook shared library dependencies.

Develop Scripts

Typically, installation tools allow the publisher to add two kinds of scripts; those executed by the tool during installation and those executed during de-installation to erase all trace of the software. Also, some software packages require customized scripts to handle special configuration during the installation process. For example, many database systems require a special userid to be added to the system.

Writing these scripts is very difficult, but many actions can be specified in a general fashion. In order to simplify the development of scripts, the publisher simply specifies the desired results of executing the script, and *mkpkg* generates all the scripts needed for the package.

Gather Components

Once the package is specified, *mkpkg* gathers all the components, such as the customization scripts and installed files, and saves them in a temporary location. In the case for which multiple versions of a single package may be built (e.g., one version for statically linked binaries and another for dynamically linked binaries), the system may generate multiple copies of the system and save them in different locations.

Assemble Package

The last step is to assemble the installation package from the package configuration, customization scripts, and saved installation. *mkpkg* creates a Product Specification File (PSF) and all the automatically generated customization scripts and then uses *swpackage* to assemble and generate the completed package.

The PSF describes all the elements, options, and content of the installation package and is used by Software Distributor during package creation. Before *mkpkg* the PSF was nearly always generated manually.

Automation

Since many of the tasks associated with building binary installation packages are structured and are common across packages, it is possible to automate most tasks. Accurate automation has the benefit of increasing the uniformity of package configuration and operation across packages.

mkpkg has automated or partially automated the following tasks:

- package manifest generation
- shared library dependency detection
- fileset and subproduct generation
- assigning files to filesets
- control script generation
- error checking

Package Manifest Generation

The first task faced by most package creators is creating a manifest or list of all the files installed as part of the package. It is critical that all files be included in the package, so it is important to reduce human error. For some packages, creating a manifest is a trivial task that can be accomplished easily by visual inspection of the software. However, packages often include dozens of files, and some packages include thousands of files. In these cases, it is very difficult to manually generate a complete and accurate manifest.

mkpkg can automatically generate a package manifest that includes all files installed as part of the software and may include some files not belonging to the package.

Shared Library Dependency Detection

mkpkg automatically detects all shared library dependencies. It checks every file in the product to discover which shared libraries are used by the product. It has a list of all shared libraries on the system and the name of the fileset that contains each library. *mkpkg* then automatically marks as a co-dependency each fileset containing a shared library needed by an executable.

When I first developed *mkpkg*, the vast majority of bugs were caused by shared library dependencies that I had overlooked. Once I added this module to *mkpkg* the number of bug reports diminished dramatically.

Fileset and Subproduct Creation

Software Distributor allows a given product to contain filesets and subproducts (groups of filesets).

Hewlett-Packard has extensive standards for fileset and subproduct naming and semantics. For example, English-language manual pages should be contained in the XXX-MAN fileset, while foreign-language manual pages should have a fileset per language (e.g., XXX-SPA-I-MAN for Spanish with an ISO character set). Fortunately, it is possible to use simple regular expression patterns to recognize when particular filesets are needed. Similarly, there is an extensive set of standards for subproduct naming based on the filesets in a product (e.g., the subproduct ManualsBy-Language always includes all filesets with non-English manual pages).

mkpkg has two ordered sets of rules for determining when to create filesets and subproducts. Each rule contains a regular expression, a threshold value, and a pattern. During fileset creation, the system iterates through the rules. It first creates a list of all files that match the regular expression. If the number of files is greater than the threshold value, then a fileset is created using the pattern (if necessary), and all the matching files are assigned to the fileset. The threshold value is used because some of the conventions are of the form "if there are enough XXX files, then put them in -YYY fileset." For example, "if there are enough manual pages, put them in a -MAN fileset."

Assigning Files to Filesets

The same rules that determine when to create filesets are used to assign files to filesets. This is particularly useful for large packages for which manual assignment of files to filesets would be tedious.

mkpkg uses the same pattern matching to decide how to assign each file to a fileset. Each file is assigned to the first fileset whose pattern matches the file name. By default all files that do not match any fileset are assigned to the -RUN fileset.

Control Script Generation

One of the most difficult tasks is developing all the control scripts that customize the remote system. Fortunately, most control scripts execute a handful of common tasks, and in many cases it is possible to automatically detect the need for these tasks.

Control scripts may be used at both the product and the fileset levels. *mkpkg* currently knows how to automate ten common tasks and allows the user to specify customization actions at either the product or the fileset level. The user specifies high-level actions that *mkpkg* maps into low-level script fragments for each of the ten possible control scripts. *mkpkg* only generates control scripts when necessary; it doesn't generate empty control files.

Error Checking

In general, it is very difficult to perform error checking for binary packages. However, there are a number of common errors that can be detected. *mkpkg* flags as many errors as possible, but there is still room for "pilot error."

Each attribute of a product, subproduct, or fileset can be marked as "required." Before assembling the package, *mkpkg* can check that every required attribute has an associated value. For example, the attribute "description" is required and *mkpkg* will generate an error if this attribute has been left blank.

mkpkg can also check that hard links do not cross fileset boundaries. In other words, if two files are joined by a hard link, then they must be in the same fileset. Optionally, *mkpkg* can ensure that symbolic links do not cross fileset boundaries.

Manifest Generation

mkpkg can automatically determine the product's manifest. In practice it is very accurate, but there are some occasional errors. There are two types of errors: excluding necessary files and including unrelated files. The most common problem is including unrelated files, and I have only had one package that did not include a necessary file.

The RPM documentation [3] states:

"RPM¹ has no way to know what binaries get installed as part of make install. There is NO way to do this. Some have suggested doing a find before and after the package install. With a multi-user system, this is unacceptable as other files may be created during a package building process that have nothing to do with the package itself."

While this comment is true, we have discovered a method which finds all files that were installed by the make install and automatically eliminates common mistakes. Human interaction provides the final check.

mkpkg creates the manifest using file timestamps to detect files that were installed by the install process included with the software source. *mkpkg* creates a new file that it will use as a timestamp, then it builds and installs the software (on the publisher's machine using the install process included with the software source). It then searches (part of) the file system for files with modification or creation times that are newer than the saved timestamp.

Since the manifest generation process uses file timestamps, it reliably detects all modified or installed files in the search region. There are two ways that *mkpkg* can miss files that should be included: directories missing from the search list, and files are that aren't installed. Sometimes, software installation tools are "too smart" and don't re-install files that have

already been installed. In this case, some files will not be installed by the tool and will not appear on the manifest. I do not have a solution to this problem.

More commonly, extra files will appear in the manifest because the files have been modified independent of the installation process. Usually, these files are log files for system events or daemon processes. In general, the list of such files is fairly static for any given machine, so we can create a list of all such files. *mkpkg* automatically eliminates most of those files by automatically removing "spurious" files from the manifest. *mkpkg* has a global list of "spurious" files that can be modified by publishers to match the active log files on their machines.

mkpkg has a default list of directories to search for new software. Currently this list includes paths from the Software Configuration Guide [4]. *mkpkg* will only search this part of the file system for newly installed files both to reduce the probability of independent user activity generating spurious file listings and to minimize the time required to search the file system for new files.

Control Script Generation

One of the most difficult tasks when developing installation packages is developing the customization scripts. These scripts are not interactive, may only rely on a restricted subset of system functionality, and must work correctly every time or they may leave the customer's system in an inconsistent state. The guidelines for writing control scripts are extensive and arcane.

Each control script is used during different phases of software installation, and there are many subtle issues regarding the roles of each script. In particular, there are some very subtle issues when software is installed on a disk shared by several computers, since, in this case, some scripts are executed only on the machine that copies the bits, while other scripts are executed on every machine that uses the software.

mkpkg provides a mechanism for automatically generating the control scripts for several customization actions, including PATH file updates, configuration file installation, removing obsolete files, adding a kernel driver or parameter, adding new users and groups, appending a fragment to a (configuration) file, and starting a daemon process. In addition, *mkpkg* can automatically detect when some of these actions are required and will automatically create the necessary customization actions.

Each customization task has a "work ticket" that specifies the type of task and any parameters. Each product and fileset has a list of these "work tickets" containing all its customization tasks.

Control Files

Table 2 shows the nine control files used by Software Distributor to customize software installations. An installation package may contain any or all of

¹Section 6.8 of [3].

these scripts. Control scripts fall into three basic categories: installation, verification, and de-installation. The installation scripts preinstall, postinstall, and configure are executed when the software is installed on the computer, and they should install and configure the software so that it may be removed safely in the future. The verification scripts checkinstall, verify, and checkremove do not perform any work. The de-installation scripts preremove, postremove, and unconfigure are designed to remove most of the customizations added by the installation phase. This is an exceptionally difficult process to perfect, especially since some customizations should not be removed because the system may now depend upon them. For example, it might be a bad idea to remove a user since the customer may have created files using that user-id.

checkinstall	executed before bits are installed to check that the software can be installed; no side-effects
preinstall	executed before bits are copied to system in preparation for installation and customization, e.g., save original versions of configuration files
postinstall	executed after bits are copied to system. Completes customizations that are shared by all systems using network bits.
configure	may be executed independently and should be executed on every system using the software. Does system-specific customizations, e.g., add-a-user.
verify	verifies that the software is properly installed and configured. No side-effects.
checkremove	executed before the bits are removed. Checks that a fileset or product may be removed.
preremove	expected before the bits are deleted. Prepares the system and the software for removal.
postremove	executed after the bits are removed. Cleans up and removes any leftover mess.
unconfigure	executed after the bits are removed. Removes (some) system-specific customizations. Not all customizations should be undone.

Table 2: Nine control files for installation and customization.

Control File Generation

When *mkpkg* is creating a product or fileset, it iterates through the work tickets to create a sequence of code fragments for each control script. If a work ticket specifies a control action in that script, then it generates a code fragment by filling in a template with

specifics from the work ticket. *mkpkg* creates each control script by concatenating the relevant code fragments.

Control Actions

The basic customization actions include: adding directories to the PATH files, adding new users or groups to the system, installing configuration files, inserting fragments to system files, removing obsolete files, adding a kernel driver or parameter, starting a daemon process and adding cron actions.

I have built over three hundred software installation packages for a wide variety of public domain applications, such as database systems, editors, compilers, and games. These software packages vary widely in many ways, but they have needed only a few types of customization. I believe that the entire body of software that I have managed needs only those customizations that have already been automated by *mkpkg*.

I was able to obtain a copy of all the control scripts written for all the software shipped by Hewlett-Packard for HP-UX using Software Distributor. I examined nearly all of the control scripts, and I think that most of the customization actions needed by Hewlett-Packard are already included in this list.

Custom Scripts

Since *mkpkg* only contains built-in customization detection and handling for a few common actions, *mkpkg* provides a mechanism for publishers to execute their own custom scripts. "Custom" scripts allow the user to specify that a given script be executed as part of a given control script phase. The given script is included in the package as a control script. *mkpkg* creates a control script fragment that executes the given script and retains the exit code.

PATH File Components

In HP-UX 10.x, there are three files /etc/PATH, /etc/MANPATH, and /etc/SHLIB_PATH that provide each user with default values for login shell environment variables. Any package that has its own directory tree would probably need to modify these files. For example, the new standard for independent software packages recommends that packages be installed under /opt/package/{bin,lib,etc,man,...}, so each package would require adding path elements to each of the path files. Fortunately, it is possible to automatically recognize that a fileset requires control script actions using filename pattern matching and file type checks.

Configuration File Installation

In HP-UX 10.x, configuration files should not be installed directly into place because end-users may modify configuration files. In addition, in a network file system environment, a package may be installed on one machine into a shared directory, and then run by each of the other machines after "configuration." Consequently, the configure scripts need to copy configuration files into an unshared location. By

convention, configuration files are contained under the `/etc/` tree, but they may be located elsewhere. Although, the system only automatically detects configuration files in some cases, *mkpkg* users may specify additional configuration files.

Since configuration files must be installed into a staging location, *mkpkg* searches for files that are slated for installation directly into the configuration area (`/etc/`), changes the installation location to the staging area (`/etc/newconfig/`), and creates a work item. For files slated for installation in the staging area, *mkpkg* creates the work item to move the configuration file into place.

New Users and Groups

Some packages require that a particular user or group own files. If it is not in the standard set of users and groups for UNIX machines, then it must be installed on the system. For example, many database systems need to run as a specific user-id, and all of the database's files are owned by that user-id. *mkpkg* can generate the control script fragments that create and remove user-ids and group-ids. In many cases, *mkpkg* can automatically detect that software requires a new user-id or group-id by examining the ownership of all the files in the product or fileset. If the software has user or group ownership by any user-id or group-id that is not in the basic set of users and groups shipped with every system, then *mkpkg* needs to create a new user-id or group-id.

mkpkg-generated control scripts do not modify system files directly, but use the system programs `useradd`, `groupadd`, `userdel`, and `groupdel` to update the system.

System File Modification

Some packages need to modify existing system files. There is a class of system files that contains system configuration information and that is relatively insensitive to the ordering or location of entries. For example, `/etc/inittab` contains a list of processes that should be started or stopped on entry and exit from various run-levels. Each entry is a single line, and the lines are position-insensitive.

mkpkg generates the control scripts that can add or remove a line (or lines) to such files. *mkpkg* needs to know the file name and line (or lines) to insert. During customization, the generated control scripts check the file for the specified lines. If they are not present, then they are appended to the system file. Decustomization may need to remove these lines from the file.

Crontab Entries

The cron system is used to execute scheduled and repetitive actions. Each user may have an individual cron schedule. cron uses a structured configuration file, called a crontab, to control its actions. The standard system file modification action would be sufficient for cron, except for the fact that the crontab

should not be modified directly. One gets a copy of the current crontab file by executing `crontab -l` and then updates the crontab by executing `crontab <crontab>` as the user whose cron schedule is being updated.

Starting a Daemon Process

Some software contains daemon processes. In many cases, the packages modify one or more system files so the daemons will be restarted automatically after a reboot. However, it is often useful to start these daemon processes immediately, without requiring a reboot. This task ensures that the daemon is started automatically on the end-user's machine during package customization.

Architecture

mkpkg has a very modular design that provides a framework for adding new modules and functionality. The user requests that *mkpkg* execute actions, such as "create the product manifest." Actions are composed of sequences of operations. The operation is the basic unit of functionality.

mkpkg executes each operation within an action in sequence. The operations may return an error code (in which case *mkpkg* may ask the packager if s/he would like to abort) and *mkpkg* adds text to the operation log. Operations have a uniform function interface. Operations are intended to function without user interaction, since *mkpkg* can be used via either a command-line interface or a GUI.

New functionality is added to the system by developing new operations, and then adding them to the appropriate action list or creating a new action.

Most of the internal structure of the system, its interaction, and user interface are all defined by data structures that specify how various pieces interact. In this way the basic code is often very simple and many pieces of the system can be reused easily and often. Sometimes the data structures are code fragments that get dynamically executed by the Tcl interpreter.

Data Structures

mkpkg's greatest weakness is its data structures for storing package configuration information; *mkpkg* uses Tcl arrays as the basic data structure container. There are two global arrays: `database` and `product`. `database` contains the system information that is used by all packages created on that system, while `product` contains the information relevant to a particular product.

The array index is a comma-separated list of defining attributes of the data value. The entire context for a given piece of information is encoded in the index. For example, the list of prerequisites for *mkpkg*'s fileset `mkpkg-BIN` is in the array element:

```
product(mkpkg,fileset,
        mkpkg-BIN,prerequisite)
```

This system is cumbersome but effective for most of *mkpkg*'s needs. As I have been developing the control script generation, its weaknesses for general hierarchical data have become more pronounced.

Operations

Operations are the basic building blocks of *mkpkg*. Each operation is atomic and may be used in many actions. Operations often modify the package or *mkpkg*'s state, but not always. For example, one action creates the timestamp file used during manifest generation, while another action builds the application. Not all actions modify *mkpkg*'s state; some are used to provide error checking.

Backends

The backends provide installation system-specific code. The backends provide two functions: *dumpPSF*, and *package*. *dumpPSF* creates the PSF file for the package using all the state and information available. *package* executes the backend-specific packaging program to create an installation package.

mkpkg is structured so that it can easily produce packages for a variety of software installation tools. In the past it has been able to create installation packages for several other installation tools.

Interfaces

There are two user interfaces for *mkpkg*: a command-line interface and a graphical user interface. The command-line interface provides access to most of the actions and functionality of *mkpkg*, but it does not have any facilities for browsing or modifying specific data fields.

The graphical interface provides access to all of the actions and functionality provided by *mkpkg*. In addition, it provides the ability to browse and edit all of the product configuration information, such as file-set manifests. Users may use the GUI to create, delete, or rename filesets and subproducts; to add or delete files from manifests; to specify customization actions; or to edit any one of the other myriad configuration items.

Advanced users may edit *mkpkg*'s data files, but this must be done with great care.

```

/usr/local/bin/less
/usr/local/bin/X11/xless
/usr/local/lib/X11/app-defaults/Xless
/usr/local/man/man1/less.1
/usr/local/man/man1/xless.1

```

Table 3: Software sources.

Developing a Package

Developing a package requires several steps; the package *less* will be used as an example. The first step is to prepare the software for packaging. We should be able to automatically compile and install the software correctly on our machine without human intervention.

Of course, if we are building a package for pre-compiled software, we can skip compilation.

The software sources are in the directory *less-1.0/*, and there is a Makefile with three targets: *all*, *install*, and *clean*. *less* contains the files listed in Table 3. Since the package is small, and since it has only a few man pages, we will ship the entire product in a single fileset *less-RUN*.

We need to decide if we will distribute code that has been statically linked or dynamically linked. In general, software that is released as part of HP-UX will be dynamically linked, while software that is shipped by third parties or is shipped independently of HP-UX may be statically linked. The advantage of static linking is that the executables do not depend on specific shared libraries and are more likely to work correctly on a wider range of platforms, but at the cost of additional disk space consumption. Our package will be shipped with dynamically linked executables. We are now ready to begin building our Software Distributor (SD-UX) product.

Secondly, we start *mkpkg* within our project directory *less-1.0*, and provide *mkpkg* with enough information to be able to build, install, and locate the software in our product. The *mkpkg* interface has a menu bar across the top. Under the 'View' menu, we can see all of the 'pages' that contain information that we may need to provide, verify, or modify. Under the 'Action' menu are all the actions that we need to produce an installation package.

We are currently viewing the 'configuration' page for the product. Notice that *mkpkg* has already provided default values for many of the attributes. Some of the defaults come from the default values on the 'system' pages, but others have been computed. For example, the product name (*less*) and version (*1.0*) have been computed from the current directory name.

On the configuration page, we need to fill in the 'directory' attribute with */usr/local*. This attribute is used in the PSF file, but it is also used by *mkpkg* during manifest generation to locate the files installed as part of the package. In some cases, we may not know where every file will be installed. *mkpkg* has a (long) list of directories in order to catch these wayward files.

We have told *mkpkg* where to look for installed files, now we need to tell it how to build and install our software. Go to the 'build' page under the 'View' menu and check the attributes 'build', 'install', and 'clean'. Their defaults 'make', 'make install', and 'make clean' are correct because they are the targets used by our Makefile.

We need to create the manifest, so select the menu option 'Create file list' on the 'Action' menu. This action may take a long time, since it compiles and installs the software, and then it searches your system for newly installed files. For large packages,

just compiling the software may take hours, while for large systems it may take hours to just search the file system for installed files. Once this action is complete, you should see a fileset 'less-RUN' and a subproduct 'Runtime' under the 'View' menu.

You should now check every attribute on each page, correcting or providing information as necessary. You should also check that the fileset 'less-RUN' contains all the files from our package (and no more!), and that the subproduct 'Runtime' contains just one fileset. Also, 'less-RUN' should have dependencies on various filesets from OS-CORE and X11.²

Thirdly, we create the installation package with the 'Create dynamic package' action. This action compiles and installs the software. It then copies the software to some 'safe' place (in the parent directory of less-1.0, there should be a directory named something like less-1.0_10.20_dynamic). It then generates a PSF file (in the 'safe' place), and uses that PSF to create an installation package. The installation package is left in the parent directory.

Experiences

I started developing software installation packages in 1993 because I wanted to provide binary installation for public domain software within Hewlett-Packard. There is a network installation tool, called ninstall, which has been in widespread use within Hewlett-Packard for a long time. I wanted to build ninstall packages for common public domain software, such as emacs, so other people inside Hewlett-Packard could install the packages and not duplicate my porting, configuration, and compilation effort.

It took me a week to generate my first ninstall package, both because I had to learn how to package software and because I had to manually create the package manifest and all the PSFs. It only took a week to write the first version of *mkpkg*, which included the automatic manifest generation and PSF generation.

I used the initial version to develop binary installation packages for about 50 packages. At this point it would take me about three minutes of effort per-package to build a complete binary installation package once the software had been ported and configured. Since *mkpkg* builds the package several times during the course of the process, the actual elapsed time can be far longer. For example, TeX took a few hours while less took a few minutes.

At this point I was supporting a library of about 50 public domain software packages which could be installed by HP employees over the HP intranet. This library was very popular and I soon had thousands of

internal "customers"; I also started getting bug reports. I discovered that my customers were having a lot of problems running programs that depended on a shared library that was missing on their machine. Usually the library was included in another package, but I had not marked the package dependency so the requisite libraries were not getting installed automatically. I then extended *mkpkg* to detect and manage shared library dependencies and the problems disappeared.

Using this version of *mkpkg*, I have been supporting over 250 packages. The biggest difficulty at this stage was developing customize/decustomize scripts for extraordinary packages. In addition, I found a few packages (e.g., TeX 3.1415) whose "make install" processes were so intelligent that the processes would only install certain files if they did not already exist. Since these files invariably existed on my machine, they were not installed during the "make install" phase of manifest generation and so they were not included in the manifest. There is no substitute for testing software packaging on a "clean" machine.

mkpkg was then extended so that it could generate both ninstall and update packages. This version of *mkpkg* was shared with the HP-UX Porting and Archive Centre so they could easily generate update packages of public domain software.

With the advent of HP-UX 10.0, update was replaced with SD-UX, the HP-UX version of Software Distributor, as the standard software installation tool. Since SD-UX added hierarchical structure on top of the simple fileset model used by update, *mkpkg* was rewritten to manage the product/subproduct/fileset structure. This hierarchy added a lot of complexity to *mkpkg*. For example, *mkpkg* now knows how to create filesets and subproducts based on standard naming conventions and other guidelines, and during manifest generation it automatically assigns files to the proper fileset. Internally, Hewlett-Packard has a variety of guidelines governing subproduct and fileset naming, assignment of files to filesets, and a myriad of other topics, and *mkpkg* tries to automate those guidelines whenever possible.

This hierarchical version of *mkpkg* was also shared with the HP-UX Porting and Archive Centre so they could start generating SD-UX packages. They have since used it to generate thousands of packages.

My final task was developing customize/decustomize scripts. While developing hundreds of ninstall packages I discovered that most packages require only a few, basic customization actions, so *mkpkg* was extended to automatically detect and generate customize/decustomize scripts for a variety of common actions.

Acknowledgements

I would like to thank Colin Charlton, Richard Lloyd, Rik Turnbull, and all of the dedicated people of the HP-UX Porting and Archive Centre who have put

²Actually, *mkpkg* will only find these shared library dependencies if you have run the action 'Search system for shared libraries' prior to running 'Create file list'.

up with pre-release versions of *mkpkg* and provided valuable feedback.

I would also like to thank Shahryar Shahsavari of Hewlett-Packard Software Integration and Distribution Organization who gave me a great deal of advice and information while I was designing the automated customization script generation. I should also like to thank David Mullaney, George Williams, Mark Mayotte, Debbie Ogden and the rest of the Software Distributor team for their support and encouragement.

Finally, I would like to thank the anonymous reviewers for the useful comments, and Gretchen Phillips for her extensive feedback.

Portability

mkpkg should be portable to other operating systems with a minimum of effort. *mkpkg* was developed on HP-UX, and it uses HP-UX specific tools for certain tasks, such as determining the shared libraries used by an executable. However, *mkpkg* has been ported to three installation tools (ninstall, update, and SD-UX), so adding another back-end for a new installation tool should not be too difficult. I have been intending to port *mkpkg* to Linux/RPM for over a year, but have not yet found the time.

I anticipate that the biggest porting problems will be caused by control script detection and generation because of operating system specific conventions and tools for many system administration functions.

Conclusions

mkpkg dramatically simplifies the process of creating installation packages, by automating most parts of the software package creation process. Using *mkpkg* a skilled user can create complex binary installation packages for Software Distributor in a few minutes of effort, a process which used to take hours or days.

Binary installation packages are very useful, but they have primarily developed and distributed by large software and operating system vendors because they are so difficult to develop. By dramatically reducing the effort and complexity associated with developing binary installation packages, it should now be possible for harried system administrators and MIS support staff to develop their own binary installation packages for software that they support and redistribute with their organization.

mkpkg is available at: http://www.hpl.hp.com/personal/Carl_Staelin/mkpkg.

Bibliography

- [1] *Managing HP-UX Software with SD-UX*. Hewlett-Packard. Part Number B2355-90080. 1995.

- [2] Edward Bailey, *Maximum RPM: Taking the Red Hat Package Manager to the Limit*, Red Hat Software, Durham, North Carolina, 1997.
- [3] *RPM HOW-TO*, <http://www.rpm.org/support/RPM-HOWTO.html>.
- [4] *Software Distribution Standard*, The HP-UX Porting and Archive Centre, <http://hpux.csc.liv.ac.uk/hppd/standard.html>.
- [5] *Standard for Information Technology – Portable Operating System Interface (POSIX) System Administration*. IEEE POSIX draft P1387.2/D13, April 1994.
- [6] Scott Hazen Mueller, *Good programs, lousy installation*. ;login: (21)3:36-38, USENIX. June 1996.

Appendix A: Glossary

- Bundle:** A collection of products and filesets that are installed as a unit by Software Distributor.
- Control script:** A script that is contained in a product or fileset and which is used by Software Distributor to check or modify the system state during software installation or de-installation.
- Customization Actions:** Actions that are not standard and that occur during software installation and de-installation. *mkpkg* customization typically modifies the system configuration so that the software runs correctly. These actions occur without user interaction.
- Dependencies:** An attribute of a package that indicates whether the package requires another package to work properly. Dependencies may be either 'prerequisite' (if the package must be installed before the current package is installed) or 'corequisite' (if the package must be installed before the current package is executed).
- Fileset:** The atomic unit of installation. Contains files and customization scripts (if applicable). May also have additional requirements, such as dependencies.
- Manifest** The list of all files to be installed.
- Product:** The primary unit of software installation in Software Distributor. It contains both subproducts and filesets and may have installation dependencies and customization actions.
- Product Specification File (PSF):** The file that describes the entire product or bundle.
- Subproduct:** A collection of filesets that are installed as a unit by Software Distributor. Subproducts are contained in products, and a product can have several subproducts. Filesets may be contained in more than one subproduct.

SEPP – Software Installation and Sharing System

Tobias Oetiker – Swiss Federal Institute of Technology, Zurich

ABSTRACT

SEPP is an application installation, sharing and packaging solution for large, decentrally managed Unix environments. SEPP can be used without making modifications to the organizational structure of the participants' servers. It provides consistent application setup, documentation, wrapper scripts and usage logging as well as version concurrency and clean software removal. This paper first gives an overview of products already available in this field and then goes on describing SEPP.

Motivation

The Swiss Federal Institute of Technology in Zurich (ETHZ) has a fairly large installation of Unix workstations. At the Department of Electrical Engineering alone, there are more than 400 workstations in operation. Most laboratories in this Department have their own system managers and file servers. The advantage of this distribution of responsibilities is that management happens close to the users. The disadvantage is that it leads to a multiplication of efforts in respect to software installation and system configuration.

In an education and research environment, the diverse user population requires a large variety of software packages in their day-to-day work. These range from little utilities to large applications taking up several GB of disk space. The IT Support Group (ISG) of our department, for example, maintains a software base of over 40 GB.

The ISG started the SEPP project with the intention to devise a software installation system which allows system managers to collaborate closely while retaining independence between the different laboratories. For the users it should bring better service by providing structured documentation on all applications, several versions of the same applications available in parallel and immediate accessibility for all applications without the need to alter `.login` or similar files.

Existing Solutions

Application installation and packaging has been an issue for years. Many solutions have been proposed and implemented. Some are mainly concerned with package installation and have no special support for networked environments:

- **The RedHat Package Manager** by RedHat Inc. [1] is the most widely used package manager in the Linux world. It is geared towards setting up software packages on stand-alone workstations, much like the SVR4 package manager. In addition to this, it provides all the

means for distributing software in source format, ready for fully automatic compilation and installation on the target system.

The Redhat Package Manager (RPM) does not impose any restrictions on package layout. Files from a package will be placed into the filesystem wherever the package author sees fit. RPM keeps track of the installed software in a database.

- **GNU Stow** by Bob Glickstein [3] is basically a link generator. The idea behind stow is to put every application into its own subdirectory tree and then generate symbolic links into `/usr/local/`. Stow's special ability is to optimize these links. If only one application provides files in `/usr/local/include`, the whole directory will be linked. Once a second package is installed which also provides files for `/usr/local/include`, stow replaces the symbolic links to a directory with a symbolic link for each include file.

In an environment with many packages this link optimization feature will not help much as most directories will be used by several packages anyway.

- **The Pack Project** by Peter Krisensen [2] is also based on the idea of installing each software package into a separate subdirectory and making the binaries available in a central bin directory using symbolic links. A special feature of Pack is that Peter Krisensen maintains a substantial public collection of 'Pack' packages at Sunsite Denmark.

Other solutions have been developed with a networked environment in mind:

- **Xhier** from the Math Faculty Computing Facility, University of Waterloo [4] is a complete software distribution and maintenance system. It provides highly automated means for compiling and distributing software in a campus setup. Xhier requires applications to be organized into packages of related software. These packages can then be distributed to a number of

workstations organized in a tree structure. To provide easy access for the user, all relevant files are linked into a common directory tree.

One major obstacle to the success of Xhier outside of University of Waterloo is that Xhier is not publicly available due to license restrictions.

- **CMU Depot** by Wallace Colyer and Walter Wong [5] is the Unix System Configuration Management component of the Workstation Administration/Host Configuration Andrew II project. CMU Depot puts an emphasis on sharing applications across the network. It was developed for an AFS environment, but works with NFS as well. Applications are organized into Collections of related software, each living in its own directory tree. Users are provided with a central directory containing symbolic links to the application binaries.
- **Depot-Lite** by John P. Rouillard and Richard B. Martin [9] is a mechanism for managing software which is designed to be light weight, easy to learn, and to provide support for multiple installed versions of a package.
- **ASIS** by Ph. Defert, E. Fernandez, M. Goossens, O. Le Moigne, A. Peyrat, I. Reguero is the Application Software Installation Server developed at CERN. ASIS is in use at CERN and other High Energy Physics Research Centers around the world. All ASIS sites work together by storing their software packages in one central repository, from where copies are made to second level servers. From there the software can either be copied to local machines or accessed directly via NSF or AFS. Installing
- **ASIS** packages is a particularly simple process by means of a GUI. Software is made available to the end-users through symbolic links written into a central binary directory. The system manager of each participating client can decide which versions of which package to install locally.
- **LUDE** by David Lebel, Duncan Fraser and Michel Dagenais [8] is the distributed software library developed at the University of Montreal. LUDE allows a distributed setup without central control. The local system manager can choose for each package if it should be run over the network or copied to the local system. Participating systems can be both client and server. Each software package is kept in a separate subdirectory. The users access packages through a binary directory from where links point to the binaries of the individual packages. Packages themselves are highly structured to allow the setup of packages which work on multiple platforms. Management of the system is performed through a single command-line tool.
- **UPS** by William Bliss, Jonathan Streets, Lourdu Udumula, and Margaret Votava is the UNIX Product Support and Distribution toolkit developed at Fermilab for the management and access of software products on local systems by the system administrators and users. UPS supports multiple concurrent versions of the same product available on the same machine. End users have to use a special setup program to prepare their account for each software package they want to use. Inter-package dependencies

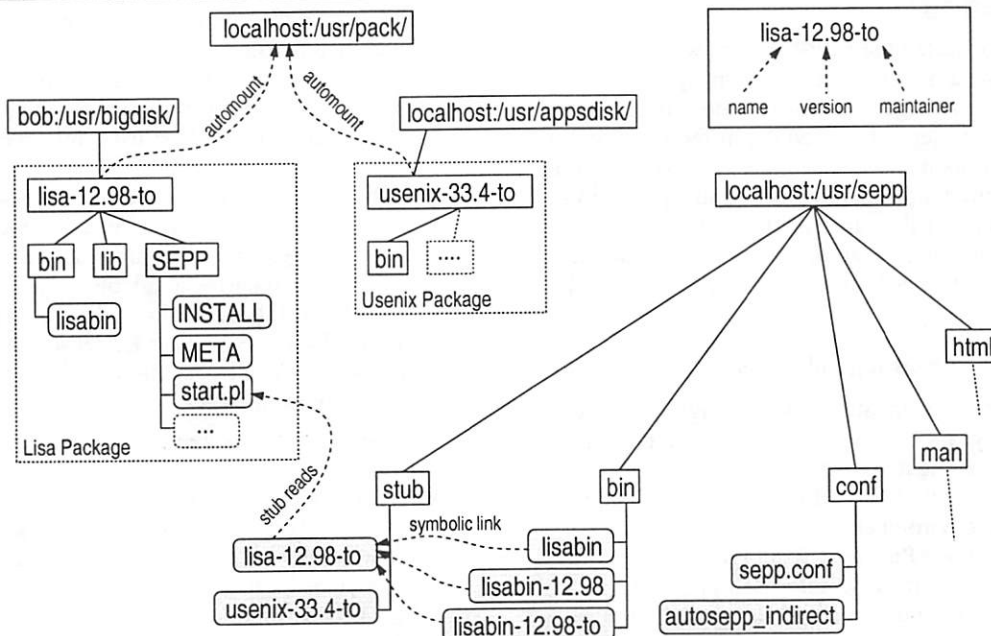


Figure 1: Components of SEPP.

are resolved automatically when running the `setup` command. Each package is assigned a status like `current`, `new`, `test`, `development` or `old`. The users can use these status labels to choose packages on a maturity level. Information about the available packages is maintained in an external database in the form of a special directory tree.

None of these packages addressed all our local requirements. Most packages were rather large compared to what we had in mind, and none supported wrapper scripts¹. A mix of features from the packages mentioned above plus some local ideas, however, provided a suitable software installation and sharing system for several Departments of the Swiss Federal Institute of Technology. We called this system SEPP.

SEPP Overview

SEPP is a package based software distribution system. Figure 1 shows the major components of a SEPP installation. Two packages are installed in this example.

- Every software package is installed into a separate subdirectory providing clean encapsulation of all files belonging to the same product.
- Every package contains a special directory called SEPP. This directory holds a few files describing the contents of the package, as well as a startup wrapper script (`start.pl`).
- This wrapper script is responsible for preparing the environment for successful execution of the binaries contained in the package. Whenever a program which was installed with SEPP is started, the program does not get executed directly, it is rather the wrapper script of the package which is called and, after preparing the environment, runs the requested program.
- Packages are made available on the local machine using the automounter. The package directories are always mounted below `/usr/pack`. This ensures that software which relies on compiled-in absolute path names finds its files.
If a package is available from several places, the automounter map is constructed to use alternate sources for the package if the primary server is not responding.
- The packages' binaries are made available to the end-users through symbolic links in the `/usr/sepp/bin` directory.
- These links do not point directly into the package directories, but to *stub* scripts stored in `/usr/sepp/stub`. SEPP generates one *stub* script for each package it installs. *Stub* scripts are written in Perl and are responsible for running the package's `/usr/pack/package/SEPP/start.pl`

file. The *stub* and the `start.pl` file together make up the wrapper script of the package mentioned above.

- Package names are built from three components:

1. The name of the package
2. The version number of the package
3. A shorthand for the name of the package maintainer

This ensures that package names are unique and everything can be mounted under `/usr/pack`.

Starting from this setup, SEPP adds many convenient features both for the users as well as for the administrators of packages.

User Features

While it is good for system managers to have a clean and well organized software setup on their servers, the user's comfort must be the prime objective. The main user-visible feature of SEPP is therefore *ease of use*:

- To use an application installed under SEPP, no changes to `.login`, `.cshrc`, or `.profile` are required, apart from adding `/usr/sepp/bin` to the `PATH` variable. The `/usr/sepp/bin` directory contains symbolic links representing all installed applications. Each program is started through a wrapper script which prepares the environment according to the requirements of the program. This includes choosing the appropriate binary in a multi-architecture environment, setting special environment variables or creating configuration files before the program is run for the first time.
- Documentation about all the locally available packages is provided on a web site and, where possible, also as manual pages. By design, SEPP forces the system manager to provide at least a minimal amount of structured documentation to be present in a package which is then used to automatically generate a documentation web site.
- SEPP supports the installation of several versions of the same package concurrently. The user can start the default version of a program by using the plain program name, while other versions are available through `program-version`. This means, for example, that Emacs 20.2 is started by using the command `emacs`. But version 19.23 is also available to users who start it with `emacs-19.23`. If two administrators are maintaining `emacs-20.2` packages and set them up differently, both packages can be installed concurrently on the same system. The versions are then distinguished by a second suffix to the executables, based on the names of the two persons maintaining the packages. The system manager of each system can decide which package and

¹Wrapper scripts are explained in the next section.

version is the default and is started by typing `emacs`.²

Management Features

The users can only benefit from SEPP's features if the system managers actually provide applications through SEPP. Therefore much effort was spent on making SEPP easy to use from the system managers point of view:

- SEPP is primarily an organizational measure. It does neither require any special daemon processes nor root privileges to work. Installing SEPP does not require altering the whole system setup. It takes only about 15 minutes to set up SEPP on a server, plus some additional time to update the clients' automounter maps and syslog configurations.
- A Perl script called `seppadm` is provided, which simplifies the maintenance of SEPP packages. The `seppadm` tool sets up skeleton installation trees for new packages and installs and removes SEPP packages from a server. Furthermore the `seppadm` tool ensures that no name clashes occur when installing a SEPP package. This is done both for stock OS binaries as well as other SEPP binaries. If clashes occur with other SEPP packages, the administrator can define whether or not the new package overrides old binaries and manual pages. Because of the elaborate naming scheme for binaries, this mechanism provides an ideal test bed setup for new versions of a package. While the previous version of the binary remains available under the normal name, the new version can be accessed by appending the version number to the binary's name.
- The automounter mounts all package directories below `/usr/pack`. This makes the physical location of a package directory irrelevant. A package can be stored on any partition of the local machine or on a remote server. The application binaries still appear to be installed under `/usr/pack/package`. This even fools setup programs of commercial applications which use `pwd` determine their installation directory.
- Every SEPP installation maintains a catalog file listing all packages stored locally, together with their NFS pathname and a short description. SEPP can be configured to use catalog files from other servers to gain access to all their

locally installed packages. This allows several SEPP servers to be tied together without requiring central management.

- A package can specify a list of other packages which are required before it can be installed. In general, however, it is preferable when a package contains all the tools and libraries it needs to run. This takes some additional disk space but is much simpler to maintain than multiple packages all cross-linked together. In our experience this policy usually does not lead to a significant growth of package size.
- The application wrapper scripts mentioned above allow the package maintainer to take any action required to make the application work, just prior to launching the program binary, without making the end-users edit their `.login` file. This cuts down support time because programs "Just Work."
- The wrapper scripts automatically log application usage through `syslog`. This enables SEPP to track application usage by configuring the `syslog` daemons on all clients to forward their messages to a central logging server.
- Some applications have configuration files which must be adjusted to the local environment. SEPP can handle this problem by copying part of the package's directory tree to `/usr/sepp/var/package` which is a local directory on every SEPP server. The application itself has to be configured to pick up its configuration file from `/usr/sepp/var/package`.

Using SEPP

The following sections give a brief example of how to create and install a SEPP package called `lisa-12.98-to` using the `seppadm` tool.

First a word on the terminology used in this section:

- **Package Preparation** is the first step to make an application available within a SEPP setup. It involves using the `seppadm` tool to create a skeleton package directory, downloading and compiling the software, installing the software into the skeleton directory and finally updating the files in the package's SEPP directory to fit the application.
- **Package Installation** makes programs contained in a SEPP package visible in the `/usr/sepp/bin` directory and therefore available to all the users who have this directory in their `PATH` variable. When installing a package, it does not matter if the package is stored on the local system or on a remote server as all file access is governed by a single automounter map. A new package only becomes visible to remote sites after it has been installed successfully on the site where it has been prepared.
- **Package Mirroring** enables the system manager to make a local copy of a package which has been installed from a remote server.

²The wrapper script mentioned above takes care of removing the version number from `ARGV[0]` so that applications which depend on being called a certain name work as expected. The wrapper also adjusts the `PATH` variable so that the program finds the correct version of any companion programs it might call during operation. For `emacs` this means that it would always use the version of `move-mail` which was installed together with the particular version of `emacs`.

Creating a SEPP Package

1. `seppadm prepare lisa-12.98-to` creates a skeleton application installation directory and updates the automounter map to make the directory available as `/usr/pack/lisa-12.98-to`. The physical location of the install directory is chosen automatically from a list of possible locations by selecting the location with the maximum available disk space. The list of storage locations has to be configured when installing the SEPP base package. It is also possible to give an absolute location when creating a package directory.
2. After downloading and unpacking the source, it can be compiled. Assuming the example package uses `autoconf`, compilation is very simple:

```
./configure \
-prefix=/usr/pack/lisa-12.98-to
make; make install
```

This configures, compiles and installs the program into the new package directory. The only change necessary to the standard compilation procedure is the use of the `prefix` argument to guide the program into the right directory and prevent it from being installed into `/usr/local/bin` where it would usually go.

3. The `seppadm` command in the first step copied

several template files into `/usr/pack/lisa-12.98-to/SEPP/`. These files must now be edited to fit the application:

- `INSTALL` contains a detailed description of the steps necessary to compile and install the package.
- `META` is a structured text file with information about the package. It includes a one-line description of the package, the addresses of the local package maintainer and support staff, and pointers to the package's binaries and documentation. The `seppadm` tool reads this file when installing a package or when regenerating the SEPP documentation web site.
- `README` is a brief description of the package. It may include information about local changes, solutions to frequent problems, ...
- `CHANGES` lists all changes which were done to the package after initial installation within SEPP.
- `patches` is a subdirectory where patches are stored which were necessary to get the package to work.
- `start.pl` is the wrapper script for the application. In the simplest case it will just

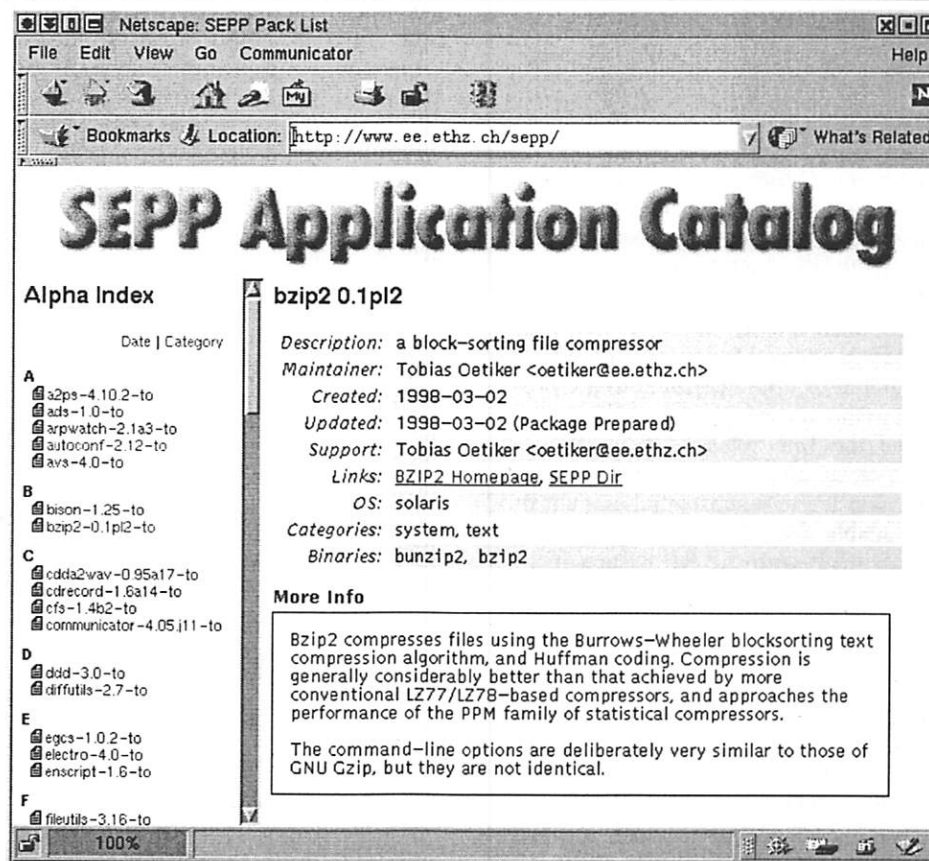


Figure 2: SEPP generated Documentation Web Site.

contain the line:

```
AppRun "bin/"
```

More problematic software products may require the setting of environment variables or the creation of per user configuration files. When a user starts an application installed under SEPP, this script will always be run before the actual application binary is executed.

Installing a SEPP Package

`seppadm install lisa-12.98-to` makes the package available on the local server. Every user who has `/usr/sepp/bin` in the `PATH` can now access the `lisabin`, `lisabin-12.98` and `lisabin-12.98-to` programs.

Mirroring a SEPP package

Remote system managers can not only install the application, they can also make a mirror copy of it, using `seppadm mirror lisa-12.98-to` to ensure maximum performance and availability.

Other Functionality of `seppadm`

Apart from the basic functions shown above, `seppadm` can also build a web site which lists all the applications installed locally (`webbuild`), as shown in Figure 2. Further, there are functions to retrieve a listing of all applications (`report`) available from remote sites, for updating the local mirrors (`mirrorupdate`) and for removing old package (`remove`).

Real World

As explained in the "Motivation" section, SEPP had to be an easy-to-use, distributed solution in order to gain acceptance within the labs. During the development phase of SEPP, the ISG kept close contact with managers from various labs of the department. An early design document was distributed to get feedback on the proposed design and features. This had the double benefit of getting people interested in the project and tidying up the design before it was even implemented. Once the first release of SEPP was available, the ISG started to install all new software under SEPP. This soon led to a substantial amount of packages being available. There was a lot of interest when talking about SEPP with the lab system managers. In most cases though it was the need to get access to some new software package from the ISG server which led to the installation of the SEPP base package on a lab server.

SEPP offers the possibility to transparently mirror a package to the local server, to enhance availability of the package as well as to reduce load on the network. Comparing the number of packages which are mirrored to the number of packages which are just cross-mounted between servers, shows that most system managers prefer to keep local copies of smaller packages while software in the `>1GB` class is

mostly cross-mounted. This might change in the future once the "switched 100 Mbit to the desktop" plan of the ETHZ is put into practice.

Future Directions

SEPP was designed to work without external databases apart from a text file listing the packages physically available on the local server and the automounter map. All other information is taken directly from the SEPP directory inside each package. With an increasing number of packages installed, the processing of this information can take a considerable amount of time. A future version of SEPP might use a cache file with pre-processed information, which has only to be updated when the `CHANGES` file of a package has been altered. Performance is not yet a problem in our setup. With 90 packages installed it takes 15 seconds to regenerate the whole documentation web site on the ISG's Ultra Enterprise 2 + SSA server. This includes analyzing all packages and writing out a web page for each one.

The current implementation of SEPP works best for user applications which are not required to successfully boot a machine. Daemon processes can be provided as SEPP packages as well, but because of the transparent automounting feature, this could lead to unintended dependencies between different servers. In the worst case, this could make it impossible to boot when two machines crash at the same time while depending on packages from each other. To prevent this problem from occurring, a feature will be added to SEPP which enforces that crucial packages are always mirrored to the local server.

With a number of sites using SEPP, it has become difficult to add major new features to the system as packages are cross mounted between different servers. One idea to alleviate this problem would be to have version numbers for the package format and make the administration script check these before installing a package. If the version number of the package format was higher than the one handled by the administration script, the system manager would be offered to retrieve a new version of the administration script.

At the Swiss Federal Institute of Technology, SPARC/Solaris is by far the most widely used Unix platform. Therefore it has been the SEPP system's primary target. The overall design of the SEPP system takes multi-platform capability into account, and it is successfully being used in a mixed Solaris/Irix environment, but running it in a really mixed environment with other than SVR4 based Unix variants would be an interesting test for the systems design.

Conclusion

The potential productivity and quality of service provided by the system managers of the department was increased both because more applications are

available to the users in a consistent setup and because the individual system managers can devote more time to direct user support and conceptual work. This in turn also leads to a better quality of life for the system administrators and thus generates a positive feedback loop.

Acknowledgments

I would like to thank my fellow system managers Elmar Heeb, Andi Karrer, Christoph Wicki, and Fritz Zaucker at the Swiss Federal Institute of Technology for the feedback during the design process and beta testing of the final product. And last but not least I would also like to extend my gratitude to Larry Wall for creating Perl.

Availability

The SEPP base package as well as details about the SEPP mailing-list are available from <http://www.ee.ethz.ch/sepp/>. SEPP is distributed under the terms of the GNU General Public License.

What is in a Name

In case you have been wondering what SEPP stands for, I must disappoint you: It is not an acronym. Sepp is a Swiss and Austrian short form for Joseph, and one might have the image of an old mountain farmer in mind when hearing the name. Maybe a future successor to SEPP will be called HEIDI.

Author Information

Tobias Oetiker got a Master's degree in Electrical Engineering from the Swiss Federal Institute of Technology, Zurich (ETHZ) in 1995. After working for one year at De Montfort University in Leicester, UK doing Unix system management, he returned to Switzerland and has since been employed by the Department of Electrical Engineering of the Swiss Federal Institute of Technology as a toolsmith and system manager.

References

- [1] RedHat Inc. *RedHat Package Manager*, <http://www.rpm.org>.
- [2] Peter Krisensen. *Pack Distribution Project*, <http://sunsite.auc.dk/pack/>.
- [3] Bob Glickstein. *GNU Stow application installer*, <http://www.gnu.ai.mit.edu/software/stow/stow.html>.
- [4] John Selens. "Software Maintenance in a Campus Environment: The Xhier Approach." *LISA V*, Sept. 30-Oct. 3, 1991.
- [5] Wallace Colyer and Walter Wong. *The CMU Depot Project*, <http://andrew2.andrew.cmu.edu/depot/>.
- [6] Anne Heavey. *UPS and UPD v4 Reference Manual*, <http://www.fnal.gov/docs/products/ups/>.
- [7] Ph. Defert, E. Fernandez, M. Goossens, O. Le Moigne, A. Peyrat, I. Reguero. *ASIS Application Software Installation Server*, <http://wwwcn.cern.pch/dci/asis/>.
- [8] David Lebel, Duncan Fraser, Michel Dagenais. *A Distributed Software Library*, <http://www.iro.umontreal.ca/lude2/>.
- [9] John P. Rouillard and Richard B. Martin. "Depot-lite: A mechanism for managing software." In *LISA VIII Proceedings*, pages 83-91, 1994.

Synctree for Single Point Installation, Upgrades, and OS Patches

John Lockard – University of Michigan
Jason Larke – ANS Communications, Inc.

ABSTRACT

The combination of large networks, frequent operating system security patches, and software updates can create a daunting task for a systems administration team. This paper presents a system created to address these challenges with system security and "uptime" as the primary concerns. By using a file-form "database," the Synctree system holds a full network's configuration in an understandable, secure, location. This paper also compares this system with previously published works.

Introduction

Synctree is flexible enough to be useful for a small organization with only a few machines, to a large university with thousands. The idea behind Synctree is that once you've got the machine talking to the network, you can "sync" it to a Synctree template to bring that system to the same level as the rest of the machines on your network. You can usually rely on your vendor to help you get to the point where your system is up and running and communicating on your network. Sun supplies Jumpstart, HP ignite, etc. But as pointed out in [Anderson], these procedures will always be inadequate for one or more reasons.

You could "clone" a system, by making an image of a system that meets your guidelines, but that image would really only be good for the initial setup of that computer, and would need to be re-generated each time a new patch was installed. For some operating systems, this may be as often as every day. In many organizations, one department will need a slightly different configuration from the next. You would also need a master machine for every system architecture you are using.

You could use rdist to send out an update, but you would need to make sure that every machine is listening when you run the update. Also, this update may only be good for one shot. Many patches require portions of previous patches to be installed first. If one of your machines wasn't "listening" when you updated with the previous patch, you can no longer count on your current patch to leave you with an operational system.

You installed a security patch on all of your systems last week, but you've just been "cracked" through the vulnerability that the patch was supposed to fix. How do you go about verifying that the patch on the system is valid? Or, are you sure that 'ifconfig' is really the 'ifconfig' that you put there? Has 'ls' been modified to pass the file size, date stamp and crc tests? The rdist timestamp/crc check is not infallible as

timestamps can be forged, and a crc on a bogus file can be adjusted by most script-kiddy "hackers" with the latest version of their favorite root-kit.

These are all challenges that we've faced in our duties as Systems' Administrators. They don't have to be as troublesome as they appear. With Synctree in place, I am able to install a patch in a central location, and be assured that the next morning all the target machines on my network will have that patch. I can also be sure that the files I expect are on my system.

The University of Michigan's Computer Aided Engineering Network group through the direction of Paul Howell created a utility called Synctree that takes care of all these things.

Previous Work

gutinteg, machdb and packagelink [Fisk] is a very similar set of utilities viewed from a completely different angle. Unfortunately, its aim is the small to medium sized operation. The focus of this idea excels at installation of packages and basic machine configuration, but is not geared toward maintaining the integrity of the system. Since the system uses a "copy" method of install rather than a "compare-before-copy" method, this can get quite expensive as more machines are thrown onto the system. Because of this, a large installation would not be able to verify the integrity of its systems on a daily basis.

GeNUAdmin [Harlander] is a very intricate system that can be used to manage even the most minute of a system's configuration. This minute focus makes this system quite daunting. The use of so many configuration "databases" leaves yet another learning curve to the use of a system that's intent is to make system administration easier. GeNUAdmin does not aim at updating of any of the system's files outside of general configuration.

Sasify [Shaddock] uses a method requiring a reboot to initiate the update to the system. For an operation that is 24/7/365, this is not a feasible method, as

many of the updates that need to be done are as simple as changing one file, with no reboot required.

OMNICONF [Hideyo] allows for the storing and restoring of configuration and operating system files. OMNICONF admittedly is useful for Updates and configuration recovery. But, the storage of whole configurations for individual machines and the lack of any real "template" sharing limits the size of the operation this can support.

lcfg [Anderson] is a very robust system for updating subsystems on a class of machine. This system focuses more on the idea of rebuilding a system from scratch when the configuration has started to "rot." Such services as Jumpstart, Kickstart and Ignite are developed specifically for the purpose of getting a machine installed, and are very adaptable to the hardware configuration of the machine. As with many other update utilities, this is also a package that runs best as a boot-time update.

Many sites use *rdist*(1) and related scripts to automate file distribution. Synctree and *rdist* share many features. However, the *rdist* model of a single fileset, which is distributed to many machines from a single server, becomes problematic in large, complex environments where universal trust for a single server may be difficult to obtain.

Remy Evard [Evard] discusses several examples of configuration management in action. He points out several problems in ad-hoc systems that we feel Synctree addresses rather well. In particular, Synctree is suitable for managing servers as well as clients (all LS&A AFS, mail, and DNS servers are Synctree clients), and uses the "configure to" model for making changes to a workstation.

Many, many other systems for performing similar tasks exist. The need for this "wheel" is so obvious, and the degree of market penetration reached by any one solution so small, that it has been reinvented in any number of places. We've been using Synctree for more than five years, and so have never looked at many of these tools. We hope that Synctree might prove useful for some of you, and also that by presenting a somewhat different spin on a common problem, we can spur other re-inventors to produce even better

work that might someday become an acknowledged standard.

What Is Synctree

Synctree is both a command and a software suite for large-scale systems administration.

As a command, it's an intelligent file copier.

As a suite, it's a whole set of programs for describing, generating, and managing system images.

Synctree, like many other packages (including *rdist*(1)), uses a compare-before-copy strategy. If a file currently on the disk is just like the file in the image, there's no need to copy. Synctree's notion of "just like" is fairly rigorous, including date, size, and md5 digest. It also checks file ownership, group, and mode, and will correct them if they are wrong but the file's contents are correct.

Since the original UNIX *cp* command doesn't work the same on all OS's, and may not work in all situations, Synctree will actually print out a list of commands needed to synchronize a file with a system image. The commands are actually shell scripts that come with Synctree, and can easily be adapted to the need or peculiarities of any given OS. They're named after their shell equivalents, with a 'sync' prepended – i.e., Synctree copies with 'synccp'. The first argument to any command is the reason why the command is being run. So, to copy down a new version of sendmail, because of a changed modification time, Synctree might say something like Listing 1. *synccp* in turn is shown in Listing 2. Along with some other refinements, the Synctree engine and these scripts were the basis of the original package.

Eventually this system breaks down. Examining a file may be simpler than copying it, but several hundred clients all checking the modification time at once on several hundred files generates severe scaling problems.

SI Files

Synctree version 2 introduced the SI database, a way of storing all the relevant information about a package. Listing 3 shows an example stanza from an SI file.

```
synccp MTIMSYNC /srv/template-server/LSA/root/usr/lib/sendmail /usr/lib/sendmail
```

Listing 1: Copying a new version of sendmail.

```
#!/bin/sh
#synccp - bourne shell script to copy a file
# Synctree Vers. 2
#grue - May 1994: initial version
#dirt - Dec 1996: replaced with cp/mv combo
cp -p $2 $3.TOBEMOVED
mv $3.TOBEMOVED $3
```

Listing 2: Using synccp.

In order, this tells us:

- What file we're comparing (*is*).
- What file to compare it to (*like*).
- What the permissions (*st_mode*) of the file should be.
- Who the owner (*st_uid*) should be.
- What group (*st_gid*) the file should belong to.
- What the file's proper size (*st_size*) is.
- What the file's *md5* digest should be.
- When and how to correct any differences (*trigger*).

The trigger value controls the sync attributes used when comparing IS and LIKE, plus the trigger specifies how a filesystem object should be instantiated. The trigger consists of three hexadecimal digits. Reading from left-to-right, the trigger semantics are as follows:

First digit – file type

- 1 Place holder, do not sync or remove.
- 2 Instantiate IS as a symbolic link using LIKE as the link text.
- 4 Instantiate IS as a regular file.
- 8 Instantiate IS as a directory.
- 9 Instantiate IS as a directory. Avoid recursing into the directory when removing extra files (e.g., /afs) but maintain the mode of the directory.

Second digit – mode,uid,gid,size

- 0 Ignore these fields for syncing purposes.
- 1 Sync the mode based on *st_mode*.
- 2 Sync the owner based on the value specified by *st_uid*.
- 4 Sync the group based on the value specified by *st_gid*.
- 8 Sync the file if the size is different from *st_size*.

Third digit – mtime,md5,remove-only

- 0 Ignore these fields for syncing purposes.
- 1 Sync the file if the time last modified is different from *st_mtime*.
- 2 Sync the file if the MD5 one-way hash is different from *md5*.
- 4 Update the time last modified and access time but do

not sync the file.

- 8 Remove IS only. No synchronization is performed. The file type specified in the first digit is used as a bit mask to specify the type of file that IS can be if it is to be removed. For example, a value of 'c08' would remove the IS if it were a regular file or a directory, but IS would not be removed if it were a symbolic link. A value of '208' would remove IS only if it were a symbolic link.

Trigger values are additive, allowing flexibility in how a file is instantiated and what attributes will trigger its replacement.

The program that creates these SI files is called *statinfo*. *Statinfo* takes a given file or directory to start with (-L, or 'like'), and a file or directory to sync (-I, or 'is'), and prints out SI files for making that comparison. For example, if I run a query like one shown in Listing 4, I'm asking "is /etc/hosts.equiv like /srv/template-server/PSC/root/etc/hosts.equiv," and I get the results shown in Listing 5.

The file specified by the -I switch doesn't have to actually exist, but the file specified by -L does, since it's the basis for the comparison.

Once the SI files are created, all the Synctree clients need to do is read them. It's much easier on the server to deliver five megs worth of SI files three hundred times than support comparison of five thousand files three hundred times.

To have an SI file, you have to have something to make it from. The standard organization method is to have a directory where all the templates live, and under each template you have a directory called "root." This directory is analogous to a machine's / directory. Any files you want to put in the template go under "root," just like they normally live under /.

Let's take a concrete example. The AFSMODS class includes four files: /usr/bin/login, /usr/X11R6/bin/xdm, /usr/sbin/in.ftpd, and /usr/sbin/in.rexecd. As a template, it looks like Listing 6.

```
% tail /srv/template-server/AFSMODS/SI/root.SI
{
is /usr/sbin/in.rexecd
like /srv/template-server/AFSMODS/root/usr/sbin/in.rexecd
st_mode 100555
st_uid 0
st_gid 0
st_size 366012
md5 b6a27adc0df35f401b54a6f9af3e342a
trigger 4f2
}
```

Listing 3: Example stanza from an SI file.

```
statinfo -I /etc/hosts.equiv -L /srv/template-server/PSC/root/etc/hosts.equiv
```

Listing 4: Similarity query.


```

/*
 * statinfo produced this output
 * executed by root on Tue Apr 14 14:18:10 1998
 * selected run-time options are:
 *     directory trigger = 870
 *     regular file trigger = 4f3
 *     symbolic link trigger = 200
 *     shadow tree trigger = 2f3
 */
{
  is /etc/hosts.equiv
  like /srv/template-server/PSC/root/etc/hosts.equiv
  st_mode 100664
  st_uid 0
  st_gid 0
  st_size 479
  st_mtime 875041276
  md5 8a0a0f460702260ba60b6098b5776887
  trigger 4f3
}

```

Listing 5: Result of similarity query.

```

% ls -laR root
root:
total 36
drwxrwxrwx  4 root    wheel    14336 Jun 24 06:25 ../
drwxr-xr-x  5 root    bin       2048 Sep  5 1996 usr/
root/usr:
total 20
drwxr-xr-x  5 root    bin       2048 Sep  5 1996 ../
drwxr-xr-x  3 jlarke  wheel    2048 Jul 12 1996 ../
drwxr-xr-x  3 root    wheel    2048 Jul 12 1996 X11R6/
drwxr-xr-x  2 root    bin       2048 Jul 15 1997 bin/
drwxr-xr-x  2 root    wheel    2048 Mar 18 1997 sbin/
root/usr/X11R6:
total 12
drwxr-xr-x  3 root    wheel    2048 Jul 12 1996 ../
drwxr-xr-x  5 root    bin       2048 Sep  5 1996 ../
drwxr-xr-x  2 root    wheel    2048 Jun  3 1997 bin/
root/usr/X11R6/bin:
total 716
drwxr-xr-x  2 root    wheel    2048 Jun  3 1997 ../
drwxr-xr-x  3 root    wheel    2048 Jul 12 1996 ../
-rwxr-xr-x  1 root    wheel   361556 Jul 25 1996 xdm*
root/usr/bin:
total 2228
drwxr-xr-x  2 root    bin       2048 Jul 15 1997 ../
drwxr-xr-x  5 root    bin       2048 Sep  5 1996 ../
-r-xr-xr-x  1 root    bin   1136292 Jun  7 1997 login*
root/usr/sbin:
total 1560
drwxr-xr-x  2 root    wheel    2048 Mar 18 1997 ../
drwxr-xr-x  5 root    bin       2048 Sep  5 1996 ../
-r-xr-xr-x  1 bin     bin     427648 Sep  5 1996 in.ftpd*
-r-xr-xr-x  1 root    wheel   366012 Mar 18 1997 in.rexecd*

```

Listing 6: Template of AFSMODS class.

Generally speaking (although this may vary in other implementations), directories on the local disk but not in a template are ignored by Synctree.

Presas and Posas

Sometimes copying a file isn't enough. For example, if we're distributing a new daemon, we want to stop an old version and start the new. Presas (pre-sync activities) and posas (post-sync activities) take care of this.

Presas and posas are keyed to a particular file and run before or after any changes to that file are made. At LS&A, we don't have much call for presas, because Synctree is able to update a running program without crashing it. We just use a posa to stop the old version and start the new after the update.

For example, Listing 7 shows an SI file entry for sendmail.

The posa line tells it to run that script after making changes to sendmail. When Synctree runs a posa

or presa, it gives it two command-line arguments: the reason why the file had to be changed, and the name of the file. So the command line might look like Listing 8, meaning that sendmail was updated because its md5 hash changed. The script is very simple; see Listing 9.

CTRL Files

Sometimes Statinfo doesn't (or can't) get all the information about a program. For example, there's no way it could tell what posa was associated with a file just by examining the file. CTRL files allow administrators to manually override all or part of an SI file. They look just like SI files, but they're usually typed by hand, and they usually have parentheses instead of curly braces. When Synctree sees a curly brace, that stanza replaces any previous stanza for that IS file. Parentheses tell it to keep the old data, and only replace the specified fields.

In the previous section, we saw an SI file for sendmail that was generated by merging an SI file and

```
{
is /usr/lib/sendmail
like /srv/template-server/LSA/root/usr/lib/sendmail
st_mode 104751
st_uid 0
st_gid 3
st_size 401640
md5 473c1c1400281a032473c1f121d0049f
trigger 4f2
posa /usr/private/admin/synctree/posas/sendmail
}
```

Listing 7: An SI file entry for sendmail.

```
/usr/private/admin/synctree/posas/sendmail MD5SYNC /usr/lib/sendmail
```

Listing 8: posa command line.

```
#!/bin/sh
# Restart sendmail if the binary was synced.
. /usr/private/admin/synctree/posas/posa.env
echo "/etc/init.d/sendmail stop"
/etc/init.d/sendmail stop
echo "/etc/init.d/sendmail start"
/etc/init.d/sendmail start
```

Listing 9: posa script.

```
{
is /usr/lib/sendmail
like /srv/template-server/LSA/root/usr/lib/sendmail
st_mode 104751
st_uid 0
st_gid 3
st_size 401640
md5 473c1c1400281a032473c1f121d0049f
trigger 4f2
}
```

Listing 10: SI file.

a CTRL file. The SI file entry would look like Listing 10, and the CTRL file entry is shown in Listing 11.

Organization (Classes)

Once you have the ability to create and use a system image, you run into the fact that no single image works for everyone. Each department, or sometimes each machine, has unique needs. So the authors created the notion of classes.

A class is a named set of files. Each department in LS&A that uses Synctree has a class. Berkeley-style printing is installed with a class, as is wuftp. We also use them to track changes to the system-one class, SOL251, is Sun's OS image with vendor patches and AFS installed. The LSA class represents all the changes we've made to suit the College's needs. If I notice that 'rdist' doesn't match the system I used to work on, I can easily find out that the LSA class replaces it with a different version.

Each class has its own SI file. Synctree reads the SI file and builds a database of files to compare and reconcile, keyed by the IS entry in the SI file.

Each machine belongs to a specific set of classes, determined by the CLASS= line in /etc/hostconfig. Every machine has its own class, which is appended to the list of classes specified in the machine's /etc/hostconfig file. The order of the classes determines which files take precedence over others. Classes later in the list win out over those listed earlier.

For example, let's say my /etc/hostconfig file includes Listing 12. Both the SOL251 class and the AFSMODS class include versions of the /bin/login program. Synctree first reads the SI file for the SOL251 class, which looks like Listing 13. and then, later on, reads the SI file for AFSMODS, which would include lines shown in Listing 14.

When Synctree sees the second entry for the same IS file, it automatically replaces the first one with the new version. When it has read all the SI and CTRL files for all the classes listed (plus the host's own class), it has a complete image of the system built up, and it's ready to start comparing and updating.

It may sound complicated, but the upshot of it is that this one program takes care of all the synchroniza-

```
(
is /usr/lib/sendmail
st_mode 104751
posa /usr/private/admin/synctree/posas/sendmail
)
```

Listing 11: CTRL file entry.

```
CLASS=SOL251:LSA:PRINT:NPI-CDDI:ODS4:AFSMODS:WUFTP242UM:PSC:PSCSOL
```

Listing 12: /etc/hostconfig line.

```
{
is /usr/bin/login
like /srv/template-server/.SOL251-sun4m_55/root/usr/bin/login
st_mode 104555
st_uid 0
st_gid 2
st_size 28800
md5 635130471cda6d64d5f2a6667dc8ecfd
trigger 4f2
}
```

Listing 13: SI file for the SOL251 class.

```
{
is /usr/bin/login
like /srv/template-server/AFSMODS/root/usr/bin/login
st_mode 104555
st_uid 0
st_gid 2
st_size 1136292
md5 3ac4528006a8ee5d0685e1e25d789673
trigger 4f2
}
```

Listing 14: SI file for AFSMODS.

tion chores – all the administrator of a machine has to do is run the command.

Installation Procedures

LS&A tries to use vendor installation procedures to install Synctree, and then let Synctree install the rest of our changes. We've only implemented this philosophy for Solaris thus far, but see no reason that it couldn't be done for other platforms.

Syncing a Machine

Just running the Synctree binary doesn't get us very far. It provides the comparison engine, but we use wrapper scripts to tell it which SI files to use, authenticate to Kerberos, and handle logging what it does.

Generally machines run Synctree from cron, starting around 3 am every morning. The crontab entry for this is shown in Listing 15. `run_synctasks` is a script which has a handy randomizer routine which will have systems sleep anywhere from 0 seconds to 6.5 minutes before starting their sync. This has been random enough for the college to avoid too large of a load all at once on the servers and the network.

If you want to run Synctree by hand, just run 'syncnode'. You can also use the '-norun' switch if you want to see what Synctree *would* do, without actually making any changes.

Making Changes to a Template

Before we change Synctree templates, we generally test the change on a particular machine first. This prevents the nightmare scenario where a change which breaks machines happens on hundreds of machines at 3 am.

Once we know that a modification works, we need to determine which class to put it in, and update that class.

Generally, we put changes in the most general template that makes sense. If we have a file that goes on every machine in a department, we don't put it in each machine's class; we put it in a class that every machine syncs to.

Updates of a class's version of a file are done with the 'ciscync' command. The basic use is just

```
ciscync <filename>
```

Ciscync will ask which class to add the file to, or we can specify it on the command line with the '-c' flag.

```
ciscync -c MATH <filename>
```

Once we've ciscync'd all the files, we need to make sure the SI files get updated to match. At LS&A, the college runs a cron job that will take care of this at

2 am each night. If you want to sync machines before then, you need to run the `mktemplate` command.

```
mktemplateSI <CLASSNAME>
```

Creating a New Class

The command to create a new class is 'mkclass'. At LS&A, running AFS, you need to be in the AFS system:administrators PTS group for the `lsa.umich.edu` cell. Class names can be no longer than eight characters long and, by convention, are in all caps. The length limit is a product of AFS' limits on volume names, since each class gets its own volume.

```
mkclass CYRUS152
```

Commonly Used Classes

For Solaris, classes of general interest at the College of LS&A include:

- [TEST,TEST4c,TEST4m,TEST4u]: Test classes for new patches, updated software, etc. If it succeeds here, it gets rolled out to production.
- [SOL251]: Stock Solaris with AFS installed, no customizations. Some files deleted in order to link them into AFS.
- [LSA]: LS&A changes to the stock Solaris install.
- [AFSMODS]: AFS-aware login, xdm, xlock, etc.
- [AFSSERVER]: Useful AFS server mods, afs fstype, tcpwrappers, etc.
- [FTP242B13]: Wu-ftpd version 2.4.2b13 with AFS mods, and .principals support.
- [MAILSERV]: Programs needed to support SMTP, IMAP2bis, POP3, and KPOP services.
- [MAILSERV2]: Like MAILSERV, but keeps inboxes in hashed directories – i.e., /var/mail/r/o/root/INBOX.
- [PERL]: Install Perl 5.004 in /usr/private/.
- [PRINT]: BSD-style printing commands.
- [SAMBA]: Samba 1.9.17p2 with AFS and NT hashed password support.

Other Synctree Utilities

Precedence

Precedence helps system administrators determine what Synctree class(es) a given file comes from. The simplest form:

```
# precedence /etc/inet/inetd.conf
```

would display a list of each /etc/inet/inetd.conf file in Synctree for that machine, from most important to least important like that shown in Listing 16 that shows that `inetd.conf` exists in MATH, LS&A and

```
0 3 * * * /usr/private/admin/synctree/bin/run_synctasks 2>&1
```

Listing 15: crontab entry for Synctree.

SOL251 and that the MATH copy is the one that takes precedence from Syntree.

mksynclist

mksynclist list Syntree clients and/or classes. Its original purpose was to provide a list of all the possible targets for mktemplateSI, but it can also be used to list all the machines that sync to a certain class.

whosyncs

whosyncs displays a list of all machines and classes that contain 'filename'

cisync

cisync copies a file from the local machine to Syntree classes. Since it uses Syntree to make the copy, administrators can be sure that the mode, type, and ownership of the file will be the same in the template, as on the local disk.

cosync

cosync gets a particular file out of Syntree. Used when you only want one or two files, and a full Syntree run would be too expensive.

addclass

addclass installs a new Syntree class on a machine.

Normally, the way to do this would be to add the class to the /etc/hostconfig file and run a syncnode. However, if you only want to add the functionality provided by the new class, there's no need to check all the files on the entire machine against the template.

rmclass

rmclass removes Syntree classes when you no longer want them on your machine.

Taking a class out of a machine's hostconfig file makes sure that the class will not sync again, but rmclass goes the extra step of removing files installed by the class.

In the interests of being careful, the output of rmclass is a list of shell commands, which you can either pipe into sh for execution or save to a file for examination.

The Big Picture: A Usage Scenario

At the University of Michigan's College of Literature, Sciences, and the Arts, Syntree and the core classes are maintained by a small group of sysadmins employed by the College. Most machine installs are done by sysadmins employed by various departments within the College. With Syntree in place, the departmental sysadmin's activities tend to revolve around it.

For example, when a new desktop machine arrives from Sun, the sysadmin uses Jumpstart to load the OS and Syntree. When the machine reboots, it has guessed (from its fully qualified domain name) which classes to install. The sysadmin double-checks this list, inputs her AFS password and the machine syncs. When it reboots, it will be a fully configured and operational workstation, already tailored to her department's needs. This operation – which can be quite tedious without automation – has taken only five to fifteen minutes of her time.

If the machine was a server, she might have added classes to install imapd and qpopper. She might have to generate a sendmail.cf file, configure DNS, or perform several other tweaks. The advantage of Syntree is that she will **not** have to worry about getting out of date versions of her mail software, an insecure version of BIND, or any of the other common bugaboos for sysadmins. She will not have to build any software from source, or get permission from a higher authority. In fact, she could deploy an entire department's worth of machines, and the expert sysadmins working for the College would never have to worry.

Comparison to Other System Configuration Systems

A detailed comparison to all systems would be impossible. However, with so much prior art in the field, a little elaboration on Syntree's unique advantages seem appropriate.

Many of the systems in common use (notably *rdist*(1)) require a server to initiate a sync. In environments where not every sysadmin trusts every other, such systems require additional automation scripts to be written in order to manage privileges. Syntree uses file system permissions to control access to the templates, and only the root user on the client machine can order a sync. Only root on the client can control which classes the client syncs to. The only parties the client needs to trust are those with administrator access to the file system the templates live on – in AFS, those in the system:administrator group.

Systems that rely on a central server also face performance bottlenecks. Even high-end machines can only carry on an *rdist* to a certain number of machines. Using *rdists*' binary comparison mode for security drops this down even further.

The only central service required by Syntree is a file server. Syntree downloads roughly five megabytes of SI files from the server, and all further work (except for copying files from AFS) happens entirely on the client. This means that much smaller demands are

```
% precedence /etc/inet/inetd.conf
/srv/template-server/MATH/root/etc/inet/inetd.conf
/srv/template-server/LSA/root/etc/inet/inetd.conf
/srv/template-server/SOL251/root/etc/inet/inetd.conf
```

Listing 16: Listing of inetd.conf files.

placed on the file server than would be placed on the rdist server. Furthermore, the use of precomputed SI files for the template, instead of forcing the server to examine each file in the template as it checks them, provides an enormous savings.

Many systems depend on complete images being distributed. Synctree, like GNU cfigengine, allows images to be overlaid by one another, to any desired resolution. As a result, a client can build up a picture in memory of the finished system before any changes are made. Using overlays without the ability to resolve conflicts in memory, "thrashes" the machine and versions of the same file are replaced by one another. For example, we once used an AFSMODS Synctree class to install an AFS Kerberos-aware login program. The default Solaris login was superceded by this version while the SI files were being merged in memory. Rdist would require two different dists, with a window of time between them when the wrong version was installed on the client.

Synctree is modular and open. The only configuration file is /etc/hostconfig, where a machine's classes are listed. Otherwise, almost all changes can be made by simply copying files. Every step of Synctree's abstraction, from the initial "I type synctree and it works" level down to watching file-by-file comparison, can be observed and understood by anyone who can read a man page and shell scripts. Other systems that hide more of their functionality, or depend on more complex grammar, can be more difficult to learn and troubleshoot.

Finally, we argue that basing the system on a file server has fundamental architectural superiorities. If the goal of a configuration system is to distribute the correct version of several hundred files to hundreds or thousands of clients, it makes sense to rely on software that has been designed for providing files on that scale. Other common concerns for configuration systems, such as security and scalability, have already been addressed and solved for file systems. Finally, many large sites already have file servers scaled to match their clients. Using this existing resource can save quite a bit of worry later on.

Reflection, Surprise, Terror, for the Future

We at the College of LS&A have been working on an update to this system that will allow Synctree to take software packages that are normally on the network, and install them on the system's local hard drive. The motivation for this, is that you usually get a new system today, where the hard drive will hold the OS around four times over. In many cases, you end up with at least a Gigabyte of space left over which may be wasted.

This idea will pull traffic off of the network and put it onto the hard drive of the local machine. By doing this, we will end up with a faster system all around. The machines with the large hard drives will

load applications locally, not off the network, and the other machines, with smaller hard-drives, will no longer have to contend with the others over that network bandwidth to get at the application.

One obviously desirable feature for Synctree would be a way to sync partial files. For example, a class for installing imapd would work best if it could also add the appropriate lines to /etc/services and /etc/inetd.conf. We've discussed doing this with either M4 macros or an implementation of the GNU cfigengine, but have yet to do any actual planning or work toward this goal.

Existing Synctree systems assume that AFS is available for file distribution. While the authors rather like AFS, we understand that not everyone has it or wants it. Synctree could easily be adapted to syncp, krcp, or any other secure client-initiated copying system. The major bottleneck is that such a system requires acls, so that clients may copy down files that are only readable by root, but cannot make changes *as* root, or (probably) grab the server's /etc/shadow file. If anyone is interested in working on this, Jason Larke may be able to provide some support.

Acknowledgments

Thanks to Rik Farrow for his support, and for prodding John into thinking that this presentation would be of interest to anyone. Also, thanks to all the UofM Unix Admins that gave the encouragement to sit down and type up this paper.

Availability

Synctree is available for non-commercial use, and is the property of the Regents of the University of Michigan, Ann Arbor, MI. Synctree (v.2) can be sampled from: <ftp://math.lsa.umich.edu/pub/Synctree/>.

Author Information

John Lockard is the Systems Administrator for the Department of Mathematics in the College of LS&A at the University of Michigan. He is thinking of switching to the Emacs paintball team. After some thinking, is really glad that Jason talked him out of titling the paper MS-SMS (Multi System-Synctree Management System). John can be reached by mail at:

Department of Mathematics
2072 East Hall - B746
525 East University Avenue
Ann Arbor, Michigan 48109-1109

He can be reached by email at jlockard@umich.edu.

Jason Larke is a Systems Administrator at ANS Communications in Ann Arbor, Michigan. He was the lead Unix Administrator for the College of LS&A previously being a member of The University of Michigan's fabled F-Five. He will never forgive himself for missing the Emacs vs Vi paintball game. Jason can be reached by mail at:

ANS CO+RE
880 Technology Drive
Ann Arbor, Michigan 48108
He can be reached by email at jarke@ans.net.

References

- [Anderson] Anderson, Paul, "Toward a High-Level Machine Configuration System," *LISA VIII Proceedings*, 1994.
- [Evard] Evard, Remy, "An Analysis of UNIX Machine Configuration," *LISA XI Proceedings*, 1997.
- [Fisk] Fisk, Michael, "Automating the Administration of Heterogeneous LANs," *Tenth USENIX System Administration Conference*, 1996.
- [Harlander] Harlander, Dr. Magnus, "Central System Administration in a Heterogeneous Unix Environment: GeNUAdmin," *LISA VIII Proceedings*, 1994.
- [Hideyo] Hideyo, Imazu, "OMNICONF – Making OS Upgrades and Disk Crash Recovery Easier," *LISA VIII Proceedings*, 1994.
- [Shaddock] Shaddock, Michael E. and Mitchell, Michael C. and Harrison, Helen E., "How to Upgrade 1500 Workstations on Saturday, and Still Have Time to Mow the Yard on Sunday," *LISA IX Proceedings*, 1995.

The Evolution of the CMD Computing Environment: A Case Study in Rapid Growth

*Lloyd Cha, Chris Motta, Syed Babar, and Mukul Agarwal – Advanced Micro Devices, Inc.
Jack Ma and Waseem Shaikh – Taos Mountain, Inc.
Istvan Marko – Volt Services Group*

ABSTRACT

Rapid growth of a computing environment presents a recurring theme of running out of resources. Meeting the challenges of building and maintaining such a system requires adapting to the ever changing needs brought on by rampant expansion. This paper discusses the evolution of our computer network from its origins in the startup company NexGen, Inc. to the current AMD California Microprocessor Division (CMD) network that we support today. We provide highlights of some of the problems we have encountered along the way, some of which were solved efficiently and others that provided lessons to be learned.

The reengineering of computer networks and system environments have been the subject of numerous papers including [Harrison92, Evard94b, Limoncelli97]. Like the others, we discuss topics related to modernization of our systems and the implementation of new technologies. However, our focus here is on the problems caused by rapid growth. With increasing requirements for more compute power and the availability of less expensive and more powerful computers, we believe that other environments are poised for rapid growth such as ours. We hope that lessons learned from our experience will better prepare other system administrators in similar situations.

Introduction

The California Microprocessor Division of AMD was formed from the merger of the Battery Powered Processors group of AMD and the newly acquired NexGen, Inc. at the end of 1995. The NexGen computing environment circa early 1995 was based primarily on Solbourne workstations and Thicknet (10base5) cabling. Over the past three years we have grown and modernized our network into our current network of nearly 1000 nodes. We have added over three terabytes of disk space and added over 500 UNIX compute servers. Our network has been through two major revolutions, incorporating technologies such as 10base5, 10baseT, ATM, CDDI, FDDI, 100baseT, and Fast EtherChannel at various points along the way. Our passwd file has grown from 433 entries at the beginning of 1996 to over 700 entries today. That number may be small by standards set by today's internet service providers (ISPs); however, the resource requirements of a typical user in our environment are far greater.

The account that follows is roughly in chronological order. We provide the reader with extended discussions on topics related to the growing pains of our compute environment.

Out of Money

Startup companies frequently have severe spending limitations, and NexGen prior to the merger with AMD was no exception. Spending authorization was handled by upper management, which had little interest or experience in dealing with large scale UNIX system environments. The early systems administration team had little contact with upper management, and was hard pressed to get approval to spend money for changes to the environment or downtime to make much needed adjustments to the network or systems.

Rectifying this situation required gathering copious amounts of information on systems issues, coming up with solutions for the problems discovered, and quantifying the loss and potential loss of revenue or schedule delays to the business as a whole. To do this analysis properly, we recruited finance and engineering management into the project, and accompanied middle managers to budget meetings to answer questions and to reinforce the business case for system improvements. The increased contact between the systems administration team and upper management led to a perception of the team as business aware, and developed a trust relationship that eased future project approval.

However, the available capital was still in short supply. Verification of x86-based microprocessors involves many many cycles of simulation to ensure

complete functionality and compatibility. The NexGen engineers were hungry for as many CPU cycles as we could provide. Our solution was to build up a farm of machines based on NexGen's own Nx586 CPU running Linux. The CPUs were obtainable with very low overhead and the peripheral hardware required was inexpensive. Demonstrating a commercial large-scale Linux installation also provided public relations benefits.

The Linux solution turned out to be short-lived. With the introduction of the UltraSparc processor, Sun was able to tilt the price-performance ratio back in their favor. In addition, the Sparc based processors were able to run almost all of our preferred CAD applications, while the Linux based machines were capable of running only a limited number of simulation programs. Fortunately, the merger of AMD and NexGen brought an immediate influx of cash which was used to purchase new machines rapidly in large quantities.

Recent developments in the microprocessor market, including the contributions of our own division, are swinging the price/performance pendulum back in the other direction. Our sister division in Austin, TX currently runs a compute farm based on AMD-K6 CPUs running Solaris x86 [AMD97]. Whether our compute farm of x86 computers was an idea ahead of its time is a subject still open to debate and is beyond the scope of this paper.

Out of Control

The early NexGen computing environment suffered from inexperienced system administration and a lack of centralized control. There were a dozen different NIS domains, partially because of an intention to separate groups of users, but also as a result of a misunderstanding of how NIS works. Since most users ended up having to have access to almost all of the domains, a conglomeration of scripts was used to manually synchronize the various NIS domains from one master domain. One of our first tasks was to merge all the domains into one.

During this period, NexGen also lacked a department responsible for deployment of CAD applications software. Such tasks were left to the whims of the individual design groups. Design engineers were forced to sacrifice valuable time installing vendor software to varying degrees of success. Applications were often installed in user home directories. Poorly written `.cshrc` files circulated among different design groups, the use of incorrect versions of tools was epidemic, and the presence of multiple copies of identical software was fairly common.

The implementation of standard "setup scripts" for CAD applications was key to improving this situation. We devised a scheme in which a single setup script would be used for each vendor's tool suite. The setup script would handle any path and environment

variable configuration needed. While crude, these scripts achieved our immediate goal of having centralized control over application use.

The users' `.cshrc` files source the setup scripts for the desired tools. An example of such a script is given in Figure 1a. A snippet of a user's `.cshrc` using this script is given in Figure 1b. `/usr/local/bin/get_arch` is a short script which returns the operating system being used - `sunos`, `solaris`, `hp9`, `hp10`, or `linux` in our example. The optional `amdpostpath` and `amdprepath` variables are used to eliminate excessive path rehashes when multiple setup files are used. If these variables are set, the user's `.cshrc` is expected to use them to build the appropriate path variable after all the desired setup scripts have been sourced. If `amdpostpath` and `amdprepath` are not set, the setup scripts will append or prepend directly to the path variable.

Some general guidelines governed the use and creation of our setup scripts:

- path and environment variables that are not vendor specific will be appended to and never overwritten.
- no assumptions should be made with regard to the order in which the setup scripts are used by users.
- the sourcing of multiple setup scripts belonging to one vendor (e.g., `setup_cadvendor_1.0` and `setup_cadvendor_1.1`) is not supported. Some checks are put in to prevent such misuse.
- the setup scripts are named by vendor and version number (e.g., `setup_cadvendor_1.0`). A symbolic link will be provided to the default version (`setup_cadvendor → setup_cadvendor_1.0`).

We could now ensure that obsolete and duplicate installations could be eliminated without leaving users in limbo. More importantly, it facilitated moves and changes to the application trees without requiring users to alter their own login files. The setup scripts allowed centralized setups for vendor tools while still allowing users freedom to customize their individual login environments. Once users had been converted to the setup script paradigm, we eliminated redundant installation of tools and performed proper reinstallation of haphazardly installed software.

There are several known limitations of our method. We currently only support `csh` based shells. There was very little demand in our user community for anything other than `csh` or `tcsh`, so we haven't been motivated to spend much effort in supporting other shells. Changes to setup scripts or tool versions are only reflected when the user's `.cshrc` file is executed. The login process is also relatively slow even with `amdpostpath` and `amdprepath` variables set. We have received a few complaints about this, but none demanding enough to make improvements to it a priority. In most cases, our users were so relieved to have a simple and relatively foolproof tool setup environ-

```

# $Id: k6system.figures.v 1.4 1998/09/28 21:02:00 lccha Exp $
# setup_cadvendor_1.0 -
#   A setup file for cadvendor
# Check for conflicts:
if ($?SETUP_CADVENDOR) then
  if ($?prompt) then
    echo "WARNING: setup_cadvendor_1.0 conflicts with"
    echo "setup_cadvendor_$$SETUP_CADVENDOR already sourced in this shell"
  endif
endif

# Find architecture of platform
if (-x /usr/local/bin/get_arch) then
  set ARCH_FOR_SETUP = '/usr/local/bin/get_arch'
else
  set ARCH_FOR_SETUP = "unknown"
endif

switch ($ARCH_FOR_SETUP)
  case 'sunos':
    if ( $?amdpostpath ) then
      set amdpostpath = ($amdpostpath /tools/cadvendor/1.0/bin)
    else
      set path = ($path /tools/cadvendor/1.0/bin)
    endif
    breaksw
  case 'solaris':
    if ( $?amdpostpath ) then
      set amdpostpath = ($amdpostpath /tools/cadvendor/1.0/bin)
    else
      set path = ($path /tools/cadvendor/1.0/bin)
    endif
    breaksw
  case 'hpux9':
  case 'hpux10':
    if ( $?amdpostpath ) then
      set amdpostpath = ($amdpostpath /tools/cadvendor/1.0/bin)
    else
      set path = ($path /tools/cadvendor/1.0/bin)
    endif
    breaksw
  case 'linux':
    exit
    breaksw
  default:
    exit
    breaksw
endsw

# Setup version variable
setenv SETUP_CADVENDOR 1.0

# Setup license files
if ( $?LM_LICENSE_FILE ) then
  if ( "$LM_LICENSE_FILE" !~ "*"1700@key"* ) then
    setenv LM_LICENSE_FILE \
      ${LM_LICENSE_FILE}:1700@keya,1700@keyb,1700@keyc
  endif
else
  setenv LM_LICENSE_FILE 1700@keya,1700@keyb,1700@keyc
endif

```

Figure 1a: Sample setup file for fictitious cadvendor (software version 1.0).

ment that they were willing to overlook these shortcomings.

We had also considered using wrapper scripts written in C-shell or Perl. In practice, we found that they take more effort to maintain when new versions of software are installed. Many of our CAD packages consist of numerous binaries which would all require individual wrappers or a link to a common wrapper. The composition of executables in the package can change frequently from version to version, forcing the system administrator installing the software to carefully inspect each release to ensure that all necessary wrappers are in place.

Some alternative methods have been presented in [Rath94], [Evvard94a], and [Furlani91]. We may look at implementing some of the ideas presented in those papers in the future. However, feedback from our users has indicated that our method is currently adequate and therefore improvements have not been a high priority.

Out of Disk Space

Prior to 1995, NexGen's data was distributed on a variety of Sun, Solbourne, and HP700 servers which were all cross mounted via NFS hard mounts to each other. This type of design results in a network with multiple points of failure, each of which can affect performance and availability of the entire network. In addition, backup of many small servers tends to be cumbersome, requiring backups running over the network or the installation of numerous local tape drives.

Early 1995 marked the arrival of NexGen's first large centralized fileserver, an Auspex NS5000. Bristling with a dozen network interfaces, it was able to provide reliable file service to all machines on the network without requiring any network router hops. Data from the various "servers" was migrated to the central fileserver and network reliability improved accordingly.

We made some decisions in the implementation of this first Auspex that we would later regret. We opted to use automount "direct" maps rather than

indirect maps based on field reports from other Auspex customers that a large quantity of indirect map entries could cause overloads on the Auspex host processor. This denied us the flexibility and scalability that indirect maps would have given us. We limited the partition size to 5GB per partition due to limitations of the backup technology we were using at the time, Exabyte 8500 8mm tape drives driven by shell scripts using dump. Each partition contained a mixture of project data, user home directories, applications, and temporary data.

The mixture of different types of directories on shared partitions combined with the relatively small size of the partitions was a nightmare to administer. Large amounts of scratch data often filled up partitions causing critical design data to be lost or corrupted. Large vendor application installations had to be awkwardly distributed among several disks.

The merger with AMD in 1996 brought a second Auspex server to our network. This gave us an immediate opportunity to revise our disk usage strategy based on our previous experience. We opted to dedicate individual partitions base on their use. We designated four general categories of disk use:

- Applications
- User home directories
- Project directories
- Critical project directories.

Partitions were individually sized based on needs. Critical project directories were allocated a sufficient amount of free buffer space to minimize disk full situations. At the opposite extreme, application directories were permitted to operate with very little free space in order to minimize waste. Access to these directories was provided by indirect maps. As of this writing we have scaled this plan to eight filesystems, each serving roughly 500 gigabytes of disk.

To monitor disk usage and to give advance warnings of partitions getting full, we wrote a disk monitoring script in Perl to be run hourly by cron. We started by using a simple script that would parse the output of the df command and generate e-mail when any disk

```
set amdpostpath
set amdprepath
foreach vendor (cadvendor mentor cadence avanti)
  if (-e /usr/local/setup/setup_${vendor}) then
    source /usr/local/setup/setup_${vendor}
  else
    if ($?prompt) then
      echo "WARNING: setup_${vendor} does not exist ... skipping"
    endif
  endif
end
set path=($amdprepath $path $amdpostpath)
```

Figure 1b: Snippet of code from user's .cshrc.

fell below a given threshold. This script did not keep any historical data and therefore was not able to detect the difference between a partition that was rapidly nearing capacity from a partition that had inched across the forbidden threshold. As a result, this script generated too many nuisance e-mails which rendered the important warnings useless.

To solve these problems, we wrote highly configurable Perl script with the following features:

- Rules based notification – rules are expressed in Perl syntax and are based on a comprehensive set of conditions tracked by the script.
- Comprehensive set of conditions – rules for notification can be based on any combination of the following conditions:
 - amount of free disk available
 - percentage of free disk available
 - time of day
 - time of last notification
 - various calendar data (month, day, year, day of week)
 - change in free disk available since last notification
 - change in free disk available since last run of script
- History database – disk usage information and a record of previous notification sent is kept. This allows the frequency of warning messages to be tuned to how quickly the disk is filling up.

Out of CPU

As product schedules became tighter and tighter, our need for faster turnaround times on our simulation and verification runs became even more critical. The brute force solution of purchasing more computers was a key part of our solution to this problem. But this alone would not allow us to reach our goal. We needed a way of using the available CPUs more efficiently. We elected to use Platform Computing's LSF (Load Sharing Facility) product to help us reach our goal of having every available CPU in use at all times.

Our model was based on having the most powerful servers located in the computer room. We deployed less powerful (i.e., less expensive) workstations or x-terminals on user desktops and encouraged users to submit all jobs, including interactive ones, to the server farm. Keeping the powerful machines off people's desktops helped prevent large jobs from causing performance problems for interactive users and reduced problems caused by "possessive" users that would complain about any background jobs running on their machines.

Our simulation jobs were typically CPU bound with minimal disk and memory requirements. Achieving maximum CPU utilization was therefore the key objective. We deployed LSF on nearly every available machine in the division as well as many machines "borrowed" from outside our division in order to

maximize the number of CPUs available. Every additional CPU we were able to utilize contributed to an improved time to market for the products being developed by our division.

LSF allows job scheduling based on several factors, including available memory, load average, and the number of LSF jobs already running on the machine. Server room machines were configured to run one job per CPU at all times. Desktop machines are configured to run batch jobs when the load average and the idle time fall within specific parameters. The actual thresholds used were tuned based on user feedback. This allowed us to get maximal use out of idle desktop CPUs while keeping the console users happy.

Further details of our configuration beyond the scope of the paper can be found in [Platform97].

Out of Space

Computers require space. Lots of computers require lots of space. Setting up workstations lined up on tables and ordinary utility shelves will work for smaller installations, but for large installations there is no substitute for well-constructed racks. For flexibility, we opted for an "open-shelf" type of rack. We used these racks to stack servers up to limits allowed by local fire codes.

Our primary goals were high density, easy accessibility, and reasonable cost. Since appearance was only a minor consideration, and access to our server room is well-controlled, we were uninterested in enclosed cabinet style racks with doors and locks. Instead, we opted for basic 23" wide racks with 11" deep ventilated shelves bolted to them front and back for an effective depth of 22". This arrangement proved to be highly versatile, accommodating PCs, Sun Sparc machines from the Sparc20s through the UltraServer2 machines, and HP 735s and J-class servers. With most of the "pizza-box" style chassis, we were able to stack up to 14 machines on 7 foot high racks. Ladder racks across the top and bolts in the floor provided stability in the event of earthquakes.

Out of Power

In August 1996, during a critical part of CMD's product development schedule, we began to notice a high number of memory failures from our compute server ranch. This led us to suspect some sort of environmental problem. We first suspected that the machines were not being adequately cooled and ventilated. After determining that this was not the cause of our maladies, we then focused on possible fluctuations in the power supply. Bingo! Our facilities department discovered that the transformer outside of our building was operating at about double the rated capacity.

The news from our utility company was not good. We could either take eight hours of downtime to get the transformer replaced immediately at the utility's expense, or risk blowing the transformer and

taking several days downtime to have it replaced at our expense. We opted to plan an orderly Saturday downtime to get the transformer replaced. In the meantime, to lessen the chance of a catastrophic transformer failure, we shutdown any equipment that was not absolutely necessary, including unused monitors, obsolete yet functional computer systems, and hallway lights. After about a week of working in a very dark building, we spent an entire weekend powering machines down and back up.

Moral of this story: check your power requirements carefully with your facilities department before it's too late. We had to pay the price with a hastily planned shutdown at a critical point in the project, but at least we were lucky enough to have not blown the transformer unexpectedly. A related requirement is to ensure that the cooling capacity of your air-conditioning system is sufficient to maintain server room temperature even on the hottest summer days.

Out of Network Bandwidth

Reaching capacity limits of our network was a persistent problem throughout our growth experience. Fortunately, as our network grew, better and faster network technology also became available. Our early network of bridges, hubs, and routers with shared

segments and multiport collision domains gave way to a completely switched network composed primarily of Xylan Omniswitches by early 1996. In subsequent years, we were able to migrate our backbone from FDDI to ATM OC-3 technology, and our end stations from 10baseT to CDDI and 100baseT interfaces.

The early network topology (circa 1995) is shown in Figure 2a. Our first major upgrade was to eliminate the hubs of shared ethernet segments and replace them with ethernet switches, creating dual-port collision domains (i.e., one machine per shared ethernet). The resulting switch based network of Figure 2b served us well for approximately three years. It suffered somewhat from the irregular growth that characterized that period of our expansion. We were several times required to add hundreds of machines to our network with no downtime permitted, which left little leeway for major topological changes. As a result, the loads across the various subnets were poorly balanced.

As the network evolved the main bottlenecks that limited our scalability were the numerous routers and the fileserver interfaces. Our new design attempted to eliminate as many of the router bottlenecks as possible. In order to do this, we attempted to flatten the network as much as possible. We had considered completely flattening the network into a single subnet, but

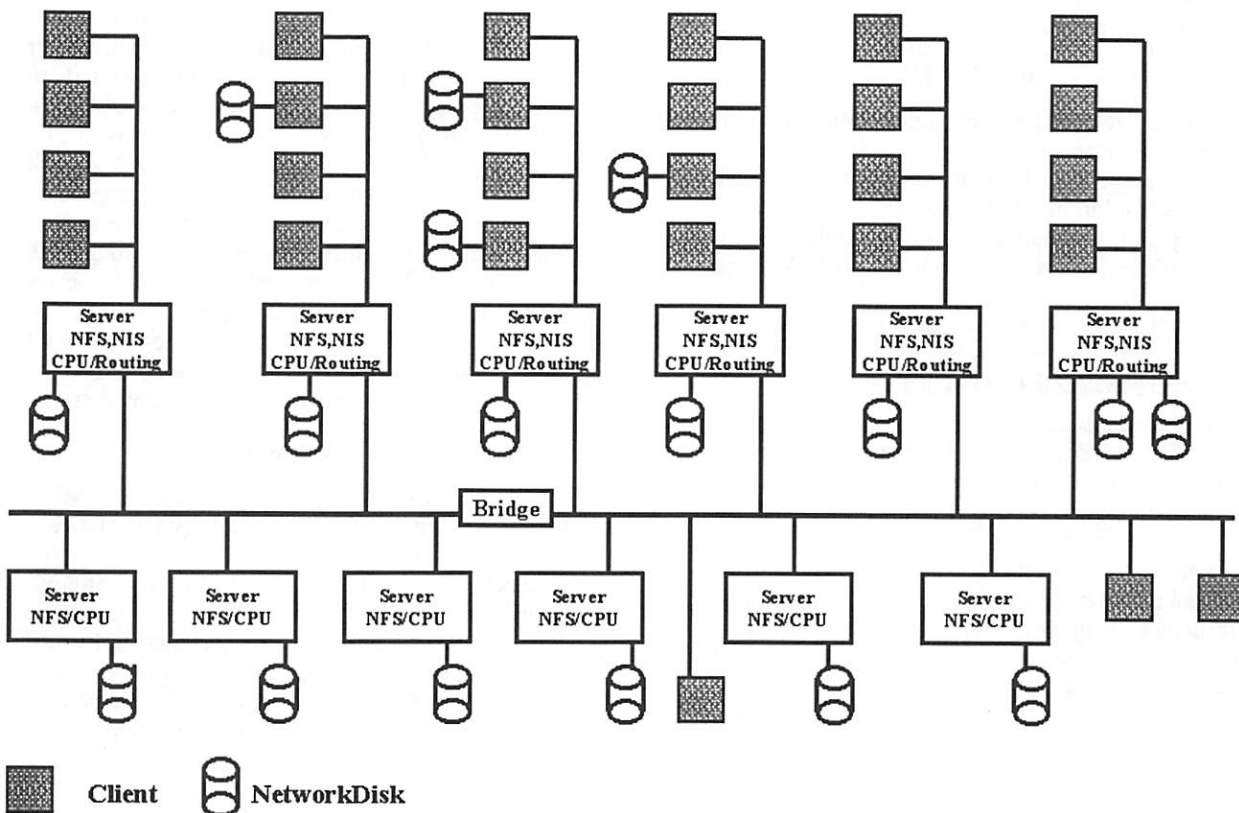


Figure 2a: Early network topology.

we were not able to come up with a suitable topology that had sufficient bandwidth at the core of the network without using unproven technology.

Providing enough throughput to the file servers would also prove to be a limiting factor in a completely flat network. We opted to use Cisco's Fast EtherChannel technology [Cisco97, Cisco98, Auspex98] to combine several full-duplex 100bT links into a single logical pipe or "channel." Currently a maximum of four links can be combined into one channel, which limits each logical interface to the file server to 800Mbps. In order to provide the desired throughput to each file server, we calculated a minimum requirement of three 800Mbps channels per file server. This implied a minimum of three subnets to avoid having the added complexity of managing a server having multiple interfaces on any one subnet.

The final design as implemented is shown in Figure 2c. The access (bottom) layer of switches provides connections to all of the client compute and desktop machines. The distribution (middle) layer consists of four Cisco Catalyst 5500 switches, each with a route-switch module to provide fast routing between subnets. These switches are responsible for traffic between the various access layer switches and all the routing between the subnets of our local network. Two Catalyst 5500 switches with router cards make up the core (top) layer, which tie together the various distribution layer switches. In addition, the core layer provides access to the routers that handle our external

network traffic. All interswitch links use Fast EtherChannel, and every switch is connected to at least two devices in the layer above it for redundancy.

Our analysis indicated that roughly 85 percent of our network traffic was NFS related. NFS traffic is also particularly sensitive to latency, so special accommodations had to be made in the topology for the NFS file servers. In our design, the file servers were connected via 800Mbps Fast EtherChannel to the various distribution layer switches to minimize the number of switch hops required by the end stations. Each file server had an interface on each of the three subnets, ensuring that every NFS client workstation had access to a file server interface without needing a router hop.

Out of Time, Part I

Size does matter [Godzilla98]. Perhaps the most valuable resource in any computing environment is the system administrator's time. Large scale computing environments highlight the need for automation. As we added more and more machines to our environment, it became obvious that many of the methods currently in place were no longer acceptable. Manual procedures had to be scripted or automated in some way. Semi-automated procedures needed to be fully-automated.

The sporadic growth of our network and the constant flurry of moves, adds and changes within our network resulted in a chaotic state of network wiring. The task of tracing a machine to its network port often

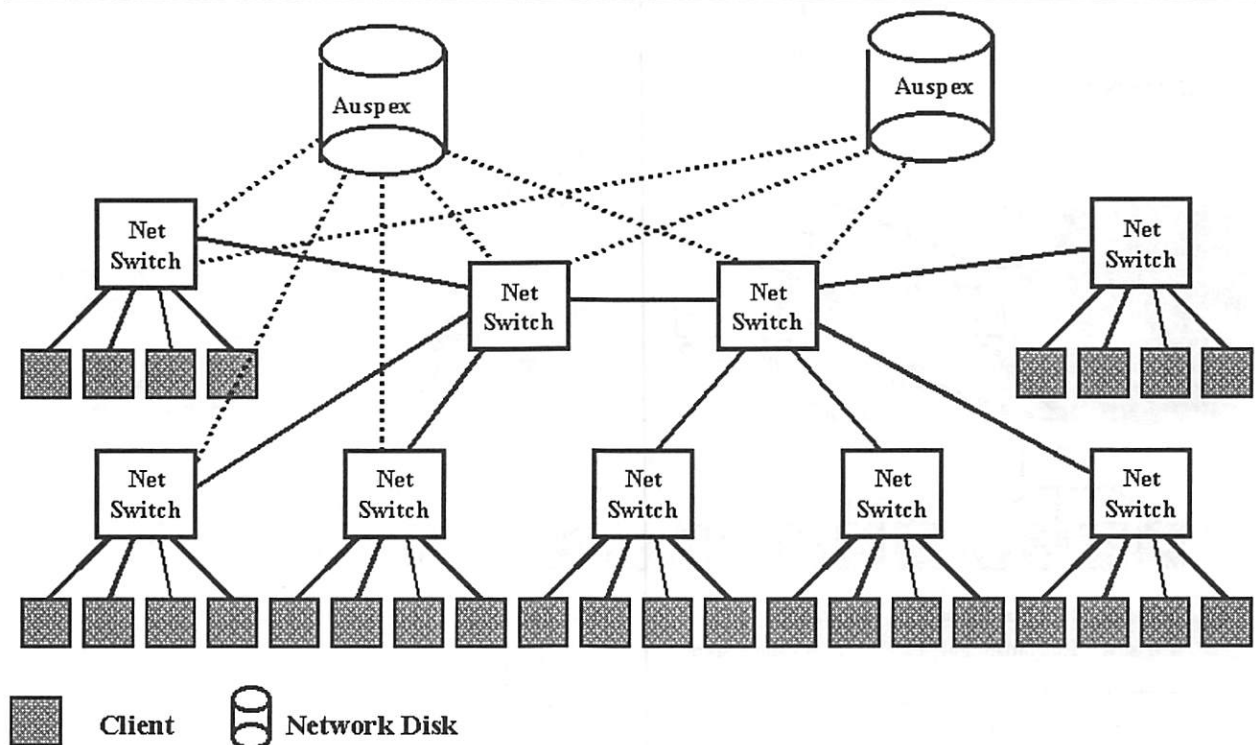


Figure 2b: Switch based network (1996).

required two people to trace through the rat's nest of cables. A week's worth of cleaning up and untangling cables was often undone in less than an hour by an emergency machine relocation or cubicle swap. To solve this problem we developed a package of scripts that collected MAC address information from the arp tables of our network switches and correlated those addresses with the information from our ethers table to accurately report where machines were connected in the network.

We realized early into our bulk purchases of workstations that we would benefit greatly by keeping machine configurations as consistent as possible. The idea was to reduce the "entropy" as described in [Evard97] thereby minimizing debug time. We had loaded our first twenty workstations by cloning hard drives using a simple script that performed a dd from one disk to another. Once the disk copy completed, a post-install script was run to take care of the machine specific configuration.

While this technique is one of the fastest methods of loading the operating systems software, it was also expensive in administrator time. Pulling the hardware off the rack, swapping in the master disk drive, starting the cloning process, and then reinstalling the system would take a minimum of 15 minutes even in best case scenarios. Updating all the machines with this method would theoretically require several man-

days and would probably require several weeks in practice.

We now employ a variety of network loading methods, Jumpstart [Sun98] for Solaris products, net-dist [HP94] and Ignite-UX [HP98] for HP-UX 9.X and 10.X respectively, and Kickstart for Linux. The basic theory in each of these is similar:

1. Perform a diskless boot using bootpd or similar protocol. A miniature version of the operating system is loaded via tftp into the swap partition
2. Load the operating system onto the local disk
3. Run a customization script to load patches and handle local configuration information.

With these methods, administrator time is now reduced to assuming control of the machine and rebooting with the designated command to force a boot from the install server rather than the local disk. The machine then takes care of the rest. Typical time to load the operating system has increased to a few hours, mostly dependent on the amount of operating system patches involved. The penalty in load time is far outweighed by the benefit of savings in administrator time.

Out of Time, Part II

Our original server room configuration had keyboards attached to every machine and a terminal that was wheeled around on a cart to attach to the console

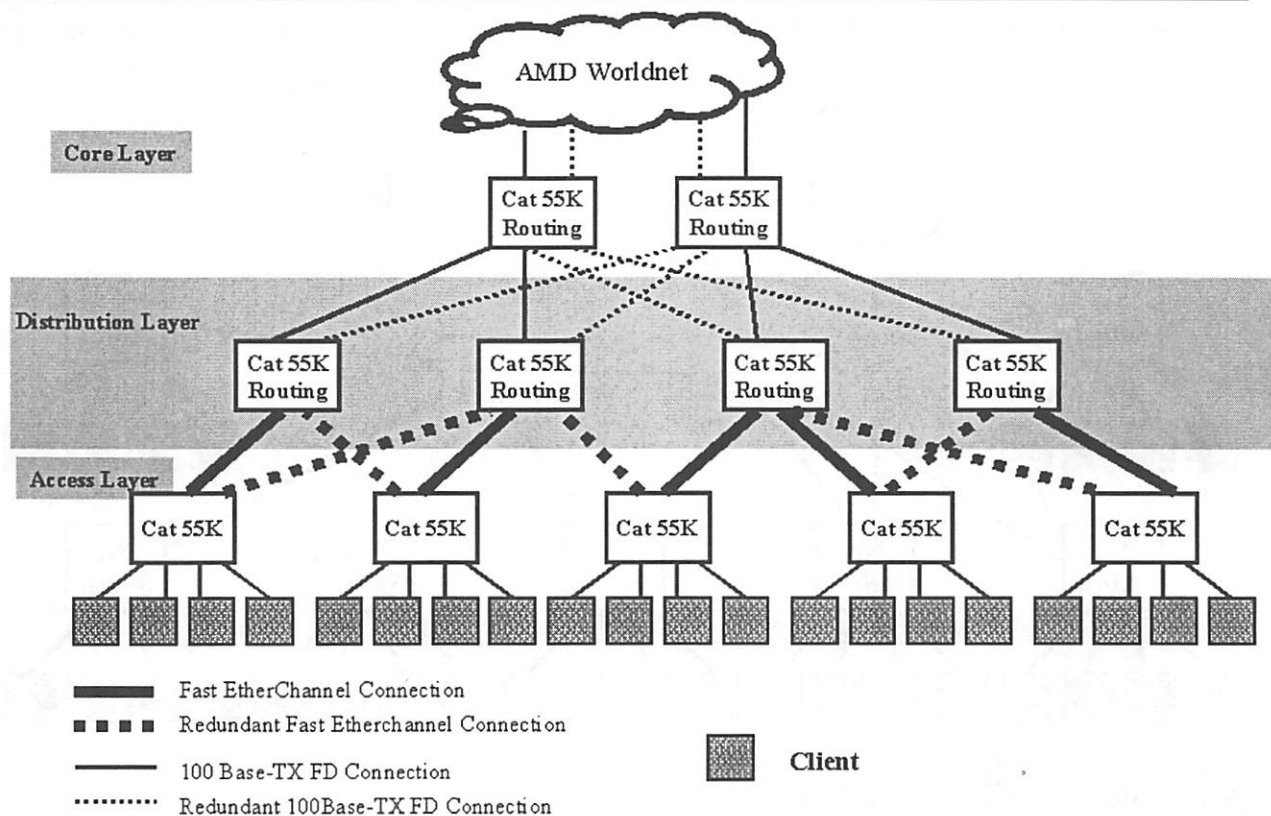


Figure 2c: Current network.

port of any machine that required attention. This was both messy and inefficient.

We were able to solve this problem by using terminal servers with "reverse telnet" capability. This feature allows one to telnet from a remote host to any of the terminal server's serial ports on the terminal server. Connecting the terminal server's serial ports to the serial console ports of the compute servers enables one to telnet directly to the console of the machine.

The default configuration of terminal servers provides access to the network from the server's serial ports. In our case, we used the terminal servers in the opposite direction, to provide access to the serial ports from the network. Consequently, many changes to the default settings of the terminal servers were required to achieve the desired functionality. We list below the most significant changes we made to our Xyplex terminal servers. Other manufacturers probably have similar options:

- **port access mode:** We set the port access mode to "remote" to allow connections to be initiated from the network side. This setting also instructs the terminal server not to output any data to the serial port other than the data being passed from the network port.
- **default session mode/telnet binary session mode:** These should both be set as passall, which directs the terminal server to pass the data from the network port to the serial port without attempting to process the data.
- **telnet echo mode:** This mode should be set to "character" to prevent the terminal server from buffering the data flow.

Our current setup uses twenty 40-port terminal servers. In order to manage the terminal servers and ensure consistent configuration we opted to boot the terminal servers from a single network boot server. This allows us to perform updates to the terminal server configurations by simply loading a new image file on the boot server and rebooting each of the terminal servers. In addition, each terminal has a local flash card that can be used for booting in the event that the boot server is unavailable.

We developed a package of shell and Expect scripts to map connections between the compute server console ports and the terminal server serial ports. Without these scripts, we would have been forced to rely on manually generated documentation. Given the frequency of server moves and changes in our environment, this documentation would have soon become outdated and useless without automatic updates.

Our Sun servers automatically use the serial port if a keyboard is not detected upon bootup. If a Sun machine receives a "break" signal on its serial console port, it halts execution and drops into ROM monitor mode. Unfortunately, power cycling of a terminal server attached to a Sun console port is often

perceived by the host as a "break" signal. [Cisco98b] suggests working around this problem by preventing the Sun computers from halting when receiving a "break."

However, this would also prevent us from halting the unresponsive machine remotely by intentionally sending a "break" signal. Since our terminal servers are protected by the same UPS system that protects our compute servers, we decided that the benefit of being able to remotely debug machines was worth the risk of leaving the servers susceptible to a global "halt" due to a power glitch.

We also ran into problems with garbage characters appearing when older telnet clients, such as those provided with SunOS 4.1.X and HP-UX 9.X, were used. We suspect that those binaries were not able to cope with some option that the Xyplex was attempting to negotiate. Compiling new telnet binaries on these machines helped eliminate some, but not all of the problems. This only affected a minority of our machines that were still running older operating systems, so we opted to ignore the problem and require that telnet connections to the consoles via the terminal server be launched from our Solaris based machines.

The Future

We are still in a state where improvements to our computing environments are bounded by a lack of time or manpower rather than a lack of ideas. We recognize that some of our solutions presented here, while adequate for our immediate needs, still leave room for improvement. Some of the projects we are currently working on are listed below:

- **Canonical hostlist project:** We have several databases that require hostname information, including the corporate DNS file, our local NIS hosts file, our local netgroup file, and various location databases. Each of these lists is currently independently maintained. Adding a host to the networks requires manually updating each of these.

We are in the process of implementing a new methodology. A single file will contain a minimum amount of data which is manually entered by the system administrator. All the rest of the information will be generated by scripts which will collect MAC addresses, network port connections, console port collections, and other data to build a master database of all host information. This database will be used to build DNS and NIS host tables, netgroup files, lists for update scripts, LSF configuration files, and documentation. The principle is to keep data entry to an absolute minimum and to have only a single source for the manually inputted data. By facilitating automated updates of the various data files, we avoid inconsistency problems.

- **Cluster monitor:** We currently monitor our systems by using scripts which process raw

data produced by syslog, LSF, and HP Open-View. LSF gives us a good overall analysis of the total throughput of our cluster. The syslog reports the hardware problems, and HP Open-View supports a variety of monitoring options. Our objective is to develop a system that will give us more detailed reporting and debugging information when there are problems. In addition, we are developing methods to allow the cluster to automatically isolate problems and perform automated fixes without human intervention.

- **Documentation:** We sometimes forget that computing environments are setup for the benefit of our users rather than for the entertainment of the system administrators. Documentation is an essential ingredient for ensuring that a computing environment can be efficiently used and is essential to prevent system administrators from being bogged down by an endless barrage of frequently asked questions. We will be making a major effort to improve our on-line documentation currently maintained on our internal web site.

Final Thoughts

We hope our experiences will be valuable in helping others plan ahead for growth in their computing installations. We have learned that it is important to solve small problems on a small scale before they expand to large problems on a large scale. Careful planning and a vision of the future is necessary to design systems that will scale easily without major renovations.

Availability

Please contact the authors at <lisa98@cmdmail.amd.com> regarding availability of scripts referenced in this paper.

Author Information

Lloyd Cha is a MTS CAD Design Engineer at Advanced Micro Devices in Sunnyvale, California. Prior to joining AMD, he was employed by Rockwell International in Newport Beach, California. He holds a BSEE from the California Institute of Technology and a MSEE from UCLA. He can be contacted by USPS mail at AMD, M/S 361, PO Box 3453, Sunnyvale, CA 94088 or by electronic mail at <lloyd.cha@amd.com> or <lloyd.cha@pobox.com>.

Chris Motta is the manager of the CMD Systems and Network Administration department. He holds a BSME from the University of California at Berkeley. He has held a variety of systems administration positions including UNIX and networking consulting. Electronic mail address is <Chris.Motta@amd.com>, and USPS mail address is M/S 366, PO Box 3453, Sunnyvale, CA 94088.

Syed Babar received his master's degree in computer engineering from Wayne State University in Detroit, Michigan. He works at Advanced Micro Devices in Sunnyvale, California as a Senior CAD Systems Engineer. He can be contacted via e-mail at <Syed.Babar@amd.com> or <Syed_Babar@hotmail.com>.

Mukul Agarwal received his MSCS from Santa Clara University. He joined NexGen, Inc. in Milpitas, California as a CAD Engineer in 1993. He switched to systems and network administration in 1995 and has been a System/Network Administrator ever since. Reach him via e-mail at <mukul.agarwal@amd.com>.

Jack Ma holds a BSCS from Tsinghua University and a MSCS from Computer Systems Engineering Institute. He was a UNIX software developer at Sun Microsystems before joining Taos Mountain at 1995, where he now works as a networking/UNIX system consultant. He can be reached electronically at <ylma@netcom.com>.

Waseem Shaikh holds a master's degree in computer engineering from the University of Southern California and received his bachelor's degree in electrical engineering from University of Engineering and Technology in Lahore, Pakistan. He was a System/Network Engineer at Steven Spielberg's Holocaust Shoah Foundation, a System Consultant at Stanford Research Institute, and is now working as a System/Network Consultant with Taos Mountain. He can be reached at <shaikh@netcom.com>.

Istvan Marko is a self-educated Computer Specialist currently working as a System Administrator employed through Volt Services Group. He can be contacted via e-mail at <imarko@pacifinet.net>.

References

- [AMD97] Unpublished internal e-mail correspondence, AMD Austin, TX, 1997.
- [Auspex98] "Auspex Support For Cisco Fast EtherChannel," *Auspex Technical Report #21*, Document 300-TC049, March 1998.
- [Cisco97] Cisco Systems, Inc. "Fast EtherChannel," *Cisco Systems Whitepaper*, 1997.
- [Cisco98] Cisco Systems, Inc. "Understanding and Designing Networks Using Fast EtherChannel," *Cisco Systems Application Note*, 1998.
- [Evard94a] Remy Evard, "Soft: A Software Environment Abstraction Mechanism," *LISA VIII Proceedings*, San Diego, CA, September 1994.
- [Evard94b] Remy Evard, "Tenwen: The Re-engineering Of A Computing Environment," *LISA VIII Proceedings*, San Diego, CA, September 1994.
- [Evard97] Remy Evard, "An Analysis of UNIX System Configuration," *LISA XI Proceedings*, San Diego, CA, October 1997.

- [Furlani91] John L Furlani, "Modules: Providing a Flexible User Environment," *LISA V Proceedings*, San Diego, CA, September 1991.
- [Godzilla98] "Godzilla," Columbia TriStar Pictures, 1998.
- [Harrison92] Harrison, Helen E, "So Many Workstations, So Little Time," *LISA VI Proceedings*, Long Beach, October 1992.
- [HP94] Hewlett-Packard Support Services, "Cold Network Installs – Configuring/Troubleshooting Guide," *Engineering Notes – Document CWA941020000*, December 12, 1994.
- [HP98] Hewlett-Packard Company, "Ignite-UX Startup Guide for System Administrators," 1998.
- [Limoncelli97] Tom Limoncelli, Tom Reingold, Ravi Narayan, and Ralph Loura, "Creating a Network for Lucent Bell Labs Research South," *LISA XI Proceedings*, San Diego, CA, October 1997.
- [Platform97] Platform Computing, "AMD's K6 Microprocessor Design Experience with LSF," *LSF News, Platform Computing*, August 1997. http://www.platform.com/content/industry_solutions/success_stories/eda_solutions/eda_stories/AMD.htm.
- [Rath94] Christopher Rath, "The BNR Standard Login (A Login Configuration Manager)," *LISA VIII Proceedings*, San Diego, CA, September 1994.
- [Sun98] Sun Microsystems, Inc. "SPARC: Installing Solaris Software," 1998.

Computer Immunology

Mark Burgess – Oslo College

ABSTRACT

Present day computer systems are fragile and unreliable. Human beings are involved in the care and repair of computer systems at every stage in their operation. This level of human involvement will be impossible to maintain in future. Biological and social systems of comparable and greater complexity have self-healing processes which are crucial to their survival. It will be necessary to mimic such systems if our future computer systems are to prosper in a complex and hostile environment. This paper describes strategies for future research and summarizes concrete measures for the present, building upon existing software systems.

Autonomous systems

We dance for our computers. Every error, every problem that has to be diagnosed schedules us to do work on the system's behalf. Whether the root cause of the errors is faulty programming or simply a lack of foresight, human intervention is required in computing systems with a regularity which borders on the embarrassing. Operating system design is about the sharing of resources amongst a set of tasks; additional tasks need to be devoted to protecting and maintaining a computer with an *immune system* so that human intervention can be minimized.

Imagine what the world would be like if humans were as helpless as computer systems. Doctors would be paged every time a person felt unwell or had to do something as basic as purge their waste 'files.' They would then have to summon the person concerned in order to perform the necessary dialysis procedures and push pills into their mouths manually. Fortunately most humans have self-correcting systems which work both proactively and retroactively to prevent such a situation from arising. Not so computers: it is as though all of our machines are permanently in hospital.

This paper is about the need for a new paradigm leading to the construction of a bona fide computer immune system. With an immune system, a computer could detect problem conditions and mobilize resources to deal with them automatically, letting the machine do the work. Although the phrase 'immune system' would make many people think immediately of computer viruses, there is much more to the business of keeping systems healthy than simply protecting them from attack by hostile programs. If one thinks of biological systems or other self-sufficient systems, such as cities and communities, some of the most critical subsystems are involved in cleaning up waste products, repairing damage and security through checking and redundancy. It would be unthinkable to do without them.

Surprisingly most system administration models which are developed and sold today are entirely based either on the idea of interaction between administrator and either user or machine; or on the cloning of existing systems. We see user graphical user interfaces of increasing complexity, allowing us to see the state of disarray with ever greater ulcer-provoking clarity, but seldom do we find any noteworthy degree of autonomy. In other words administrators are being placed more and more in the role of janitors or doctors with pagers. We are giving humans more work, not less.

The aim here is to promote serious discussion and research activity in the area of autonomic system maintenance. System administration overlaps with so many other areas of computing that it is generally forgotten as a side issue by the academic community. I would like to argue that it is one of the most pressing issues that we face. Dealing with the complexity of the network is the main challenge of the next century. Every multiprocess computer system is already a micro-cosmic virtual network. Computer resources have perhaps been too precious to make defensive or preventative systems feasible before now (and we have been distracted by other more glamorous issues), but the time is right to build not merely fault tolerant systems, but self-maintaining, fault-corrective systems. In the sections which follow I would like to explore this idea and discuss how one might efficiently build such systems.

Historical

The idea of self maintaining computer systems is not new but, as with many modern technologies (telecommunications, robotics), it originates in science fiction rather than science fact. There are dozens of examples of autonomous systems in speculative writings. The artificial intelligence community has been developing analogous systems using techniques developed over the past thirty years; some of these have even been used to create diagnostic systems for human beings, but not computers.

In 1974, science fiction writer John Brunner wrote *Shockwave Rider* [1], building on Alvin Toffler's *Future shock*. In his world of fax machines, laser printers, laptops and mobile phones, where governments argue about the public freedom to encrypt data, we find computer worms which propagate across the equivalent of the Internet performing vital (and non-vital) services quite autonomously. In this world, most computing transactions occur by creating worms, or intelligent agents which work in the background on behalf of users. Such is the extent of these worms that operating systems are necessarily programmed to give them a low priority to avoid being swamped (spammed). This is something which we experience today. His solution is correct but too simplistic for a real world system. A full immune system would need to be less passive. It was, incidentally, only a few years later in 1988 that the first Internet worm (propagating infectious agent) was thrust to the forefront of our attention [2]. Even earlier examples of autonomous systems include Robbie the robot in *Forbidden Planet* and HAL in the film *2001: A space odyssey*. HAL was a self diagnosing, but not self-repairing system, but he was also guilty of mobilizing human power and even sending them on wild goose chases, to fix a problem which could almost certainly have been dealt with automatically!

There are several valuable insights to be made by comparing computing systems to biological and social systems. Biological and social systems have solved most of the problems of self-sufficiency with ingenious efficiency. Science fiction writers too have expended many pages exploring the consequences of speculative ideas. It is not merely for whimsical amusement that such comparisons are valuable. All ideas should be considered carefully, particularly when they are based on millions of years of evolution or a hundred years of reflection.

Mechanical robots manage removable storage media even today. Robots which repair computer hardware are experimented with in England, but software robots – artificial agents which perform manual labour at the system level – are almost non-existent. One exception is cfengine [3, 4, 5, 6], a software robot which can sense aspects of the state of the system and alter its program accordingly. Cfengine can perform rudimentary maintenance on files and processes, but it is at the lower threshold of intelligence on the evolutionary scale. A system like cfengine will be the hands or manipulators of our future systems, but more complex recognition systems are needed to select the best course of action. Cfengine is not so much as robot as it is a claw.

One of science-fiction's common scenarios is that machines will run amok and turn against humankind. In a sense, bug ridden software does just this today, and system crackers write programs which corrupt the behavior of the system so as to attack the user. Isaac Asimov's answer to this problem, developed in

detail in the 1940's, was to endow automatons with a set of rules which curbed their behavior and prevented them from harming humans. In a sense this was a theoretical immune system.

1. *A robot may not injure a human being, or through inaction allow a human being to come to harm.*
2. *A robot must obey the orders given to it by human beings, except where such orders would conflict with the First Law.*
3. *A robot must protect its own existence as long as such protection does not conflict with the First or Second Law.*

These rules are more than nostalgia; there is a serious side to building the analogue of Asimov's laws of robotics [7] into operating systems. If one replaces the word robot with system and human with user, it seems less fanciful. The practical difficulty is to translate whimsical words into concrete detectable states. This starts to sound like artificial intelligence, but a less intensive solution might also be possible. In fact, such basic rules already work as a loose umbrella for the way in which systems work, but that looseness can be tightened up and made into a formal protocol. As Asimov discussed in the forties, the potential for human abuse of systems which are required to follow rigid programs is great. The system vandals of the future will have new rule systems to exploit in their pursuit of mischief. Our task is to make the rules and protocols of the future as immune as possible to corruption. This is only possible if those rules present a moving target, i.e., we aim for adaptive systems.

As a chemist, Asimov based his robots on analogue computing technology with varying potentials, not unlike the behavior of the body. In modern jargon they were based on fuzzy logic. Digital systems abandoned analogue computing long ago, but there is still a statistical truth in such analogue notions. Continuous variables may yet replace the digital logic of our canonical programming paradigms in a wide range of applications, not as analogue electrical potentials but as statistical or thermodynamical average potentials. Quantities analogous to physical variables temperature and entropy can be defined on the basis of the average behavior of computer systems. Such variables act as book keeping parameters and could be used to simplify and make running sense of system logs, for example. In his *Foundation* books, a statistical theory of society called *psycho-history* was proposed. The reality of this may be observed in the present day statistical mechanics of complex physical and biological systems (including immunology) as well as in weather forecasting. The statistical analysis of complex systems in the natural world is a science which is presently being constructed from origins in statistical physics. It will most likely converge with the work in pattern recognition and neural computing. Computer immunology needs to be there alongside its biological counterpart.

A lot of research work has been devoted to the development of mechanical robots, in the areas of pattern recognition and expert systems, but at the bottom of all of these lies a computer system which makes humans subservient to its failures.

Present Day Solutions

Present day computer systems are not designed with any sophisticated notions of immunity in mind, but most of them are flexible enough to admit the integration of new systems. How far could we go in constructing an immune system today, even as an afterthought? Many proponents of automation have built systems which solve specific problems. Can these systems be combined into a useful cooperative? The LISA conferences have reported many ideas for automating system administration [8, 9, 10, 11]. Most of these have been ways of generating shell or perl scripts. Some provide ways of cloning machines by distributing files and binaries from a central repository.

Cfengine on the other hand is a tool, written by the author, which differs from previous systems in a number of ways. Firstly it does not use linear, procedural programming such as shell or perl, it is a much higher level descriptive language. The second difference is that it has converging semantics, i.e., one describes what a system should look like, and when the system has been brought to that state, cfengine becomes inert. A third point about cfengine is that its decision making process is based on abstract classes which allows for more powerful administration models than we have traditionally been used to. Finally it offers protection against unfortunate repetition of tasks and hanging processes in situations where several administrators are working independently with little opportunity to communicate [12]. Cfengine was designed with computer immunity in mind.

In spite of the enormous creative effort spent developing the above systems, few if any of them will survive in their present form in the future. As indicated by Evard in a presentation at LISA 1997 [13] analyzing many case studies, what is needed now is a greater level of abstraction. Although its details are not yet optimal, the idea behind cfengine is basically sound and meets most of Evard's requirements, but even this will not survive in present form. It is built as a patch for our present operating systems. Ideally such a system would be built into the core of a modern operating system. The present Unix model is in need of an overhaul: even a small one would help significantly.

Corrective systems are not the only way in which one can improve present day computers. Network services are a mixture of uncoordinated mechanisms, using `inetd` or `listen` to start heavyweight processes, or based on permanently listening daemons. An interesting model which could replace these tools

is the ACE system [14]. ACE (the Adaptive Communication Environment) is an extensive base of C++ classes which provide the necessary paradigms for network services in neatly packaged objects. ACE can use lightweight processes (threads) or heavyweight processes, and can load classes on the fly in order to optimize the servicing of network protocols. ACE is well structured and carefully crafted, even though it attempts to straddle and conceal the differences between diverse Unix systems and NT. This kind of modular approach could be used to strengthen network reliability and security.

Many projects now under development, could help to improve the state of off-the-shelf operating systems. It will be up to system designers to adopt these as standards. The challenge is to compress a protective scheme into low overhead threads which will not noticeably affect system performance during peak usage. The intervention of a human should be as far as possible avoided.

Management Tools

The main focus in system administration today is in the development of man-machine interfaces for system management.

Tivoli [15] is a Local Area Network (LAN) management tool based on CORBA and X/Open standards; it is a commercial product, advertised as a complete management system to aid in both the logistics of network management and an array of configuration issues. As with most commercial system administration tools, it addressed the problems of system administration from the viewpoint of the business community, rather than the engineering or scientific community. It encourages the use of IBM's range of products and systems, and addresses other widely used systems through its use of open standards. Tivoli's most important feature in the present perspective is that it admits bidirectional communication between the various elements of a management system. In other words, feedback methods could be developed using this system. The apparent drawback of the system is its focus on application level software rather than core system integrity. Also it lacks abstraction methods for coping with real world variation in system setup.

HP OpenView [16] is a commercial product based on SNMP network control protocols. OpenView aims to provide a common configuration management system for printers, network devices, Windows and HP-UX systems. From a central location, configuration data may be sent over the local area network using the SNMP protocol. The advantage of OpenView is a consistent approach to the management of network services; its principal disadvantage, in the opinion of the author, is that the use of network communication opens the system to possible attack from hacker activity. Moreover, the communication is only used to alert a central administrator about perceived problems. No

automatic repair is performed and thus the human administrator is simply overworked by the system.

Sun's Solstice [17] system is a series of shell scripts with a graphical user interface which assists the administrator of a centralized LAN, consisting of Solaris machines, to initially configure the sharing of printers, disks and other network resources. The system is basically old in concept, but it is moving towards the ideas in HP OpenView.

Host Factory [18] is a third party software system, using a database combined with a revision control system [19] which keeps master versions of files for the purpose of distribution across a LAN. Host Factory attempts to keep track of changes in individual systems using a method of revision control. A typical Unix system might consist of thousands of files comprising software and data. All of the files (except for user data) are registered in a database and given a version number. If a host deviates from its registered version, then replacement files can be copied from the database. This behavior hints at the idea of an immune system, but the heavy handed replacement of files with preconditioned images lacks the subtlety required to be flexible and effective in real networks. The blanket copying of files from a master source can often be a dangerous procedure. Host Factory could conceivably be combined with Cfengine in order to simplify a number of the practical tasks associated with system configuration and introduce more subtlety into the way changes are made (it is not always necessary to replace an arm in order to remove a wart). Currently Host Factory uses shell and Perl scripts to customize master files where they cannot be used as direct images. Although this limited amount of customization is possible, Host Factory remains essentially an elaborate cloning system.

In recent years, the GNU/Linux community has been engaged in an effort to make Linux (indeed Unix) more user-friendly by developing any number of graphical user interfaces for the system administrator and user alike. These tools offer no particular innovation other than the novelty of a more attractive work environment. Most of the tools are aimed at configuring a single stand-alone host, perhaps attached to a network. Recently, two projects have been initiated to tackle clusters of Linux workstations [20, 21].

While all of the above tools fulfill a particular niche in the system administration market, they are basically primitive one-off configuration tools, which lack continuous monitoring of the configuration. It would be interesting to see how each of these systems handled the intervention of an inexperienced system administrator who, in ignorance of the costly software license, meddled with the system configuration by hand. Would the sudden deviation from the system model lead to incorrect assumptions on the part of the management systems? Would the intervention destroy the ability of the systems to repair the condition, or

would they simply fail to notice the error? In most cases, it is likely that all three would be the result. The lack of continuous assessment is a significant weakness.

Monitoring Tools

Monitoring tools have been in proliferation for a number of years [22, 23]. They usually work by having a daemon collect some basic auditing information, setting a limit on a given parameter and raising an alarm if the value exceeds acceptable parameters. Alarms might be sent by mail, they might be routed to a GUI display or they may even be routed to a system admin's pager [23].

The network monitoring school has done a substantial amount of work in perfecting techniques for the capture and decoding of network protocols. Programs such as *etherfind*, *snoop*, *tcpdump* and *Bro* [24] as well as commercial solutions such as *Network Flight Recorder* [25] place computers in 'promiscuous mode' allowing them to follow the passing data-stream closely. The thrust of the effort here has been in collecting data, rather than analyzing them in any depth. The monitoring school advocates storing the huge amounts of data on removable media such as CD to be examined by humans at a later date if attacks should be uncovered. The analysis of data is not a task for humans however. The level of detail is more than any human can digest and the rate of its production and the attention span and continuity required are inhuman. Rather we should be looking at ways in which machine analysis and pattern detection could be employed to perform this analysis – and not merely after the fact. In the future adaptive neural nets and semantic detection will be used to analyze these logs in real time, avoiding the need to even store the data.

An immune system needs to be cognizant of its local host's current situation and of its recent history; it must be an expert in intrusion detection. Unfortunately there is currently no way of capturing the details of every action performed by the local host, in a manner analogous to promiscuous network monitoring. The best one can do currently is to watch system logs for conspicuous error messages. Programs like *SWATCH* [23] perform this task. Another approach which we have been experimenting with at Oslo college is the analysis of system logs at a statistical level. Rather than looking for individual occurrences of log message, one looks for patterns of logging behavior. The idea is that, logging behavior reflects (albeit imperfectly) the state of the host [26].

Fault Tolerance and Redundancy

Fault tolerance, or the ability of systems to cope with and recover from errors automatically, plays a special role in mission critical systems and large installations, but it is not a common feature of desktop machines. Unix is not intrinsically tolerant, nor is NT, though tools like *cfengine* go some way to making

them so. In order to be fault tolerant a system must catch exceptions or perform preliminary work to avoid fault occurrence completely. Ultimately real fault tolerance must be orchestrated as a design feature: no operation must be so dependent on a particular event that the system will fail if it does not occur as expected.

One of the reasons why large social and biological systems are immune to failure is that they possess an inbuilt parallelism or redundancy. If we scrape away a few skin cells, there are more to back up the missing cells. If we lose a kidney, there is always another one. If a bus breaks down in a city, another will come to take its place: the flow of public transport continues. The crucial cells in our bodies die at a frightening rate, but we continue to live and function as others take over. component is very important.

Fault tolerance can be found in a few distributed system components [27]: in file-systems like AFS and DFS [28]. Disk replication and caching assures that a backup will always be available. RAID strategies also provide valuable protection for secondary storage [29]. At the process level one has concepts such as multi-threading and load balancing. Experimental operating systems such as Plan 9 [30] and Amoeba [31, 32] are designed to be resistant to the performance of a single host by distributing processes transparently between many cooperating hosts in a seamless fashion. Fault tolerance in Arjuna [33] and Corba [34] is secured in a similar way.

Ideally however we do not want fault tolerant systems but systems which can correct faults once

they have occurred. Faults are inevitable: they are something to be embraced, not swept under the rug. Some work has been done in this area in order to develop software reliability checks [35], but the reliability of an entire operating system relies not only on individual software quality but also on the evolution and the present condition of the system in its entirety. It is impossible to deal with every problem in advance. Presently computer systems are designed and built in captivity and then thrown, ill-prepared, into the wild.

Feedback Mechanisms: cfengine

Cfengine [3, 4, 36] fulfills two roles in the scheme of automation. On the one hand it is an immediate tool for building expert systems to deal with large scale configuration, steered and controlled by humans. It simplifies a very immediate problem, namely how to fix the configuration of large numbers of systems on a heterogeneous network with an arbitrary amount of variety in the configuration. On the other hand, cfengine is also a significant component in the proposed immunity scheme. It is a phagocyte which can perform garbage collection; it is a drone which can repair damage and build systematic structures.

A reactor, or event loop, is a system which detects a certain condition or signal and activates a response. Reactor technology has penetrated nearly all of the major systems on which our networks are based. It is at the center of the client-server model, and windowing technology. It is a method of making decisions in a dynamical and structured fashion. Reactors must play a central role in computer immunity.

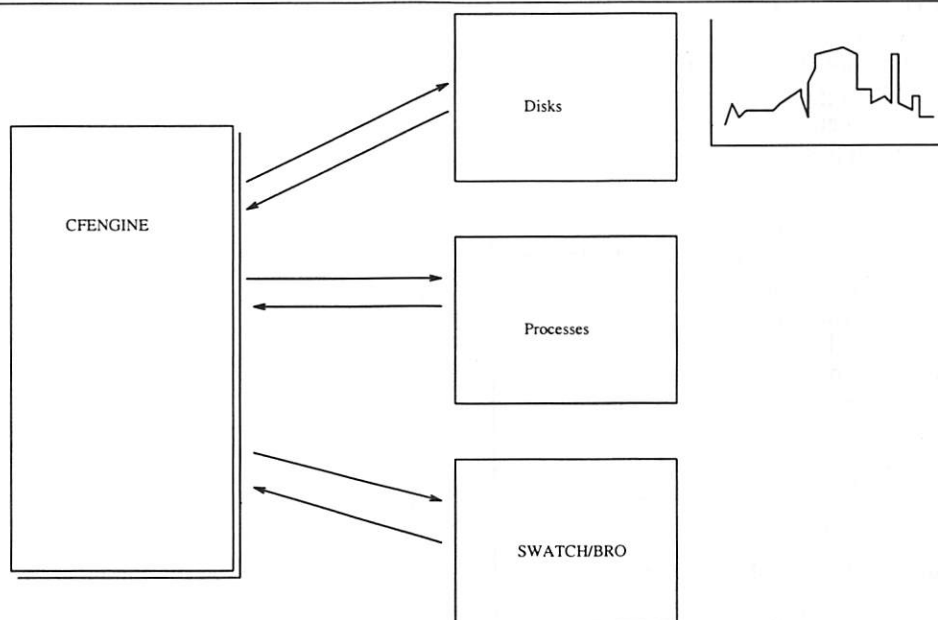


Figure 1: Cfengine communicates with its environment in order to stabilize the system. This communication is essential to cfengine's philosophy of converging behavior. Once the system is in the desired state, cfengine becomes quiescent.

Several systems are already based on this idea. Cfengine is a reactor which works by examining the state of a distributed computer network and switching on predefined responses, designed to correct specific problems. SWATCH [23] is a reactor which looks for certain messages in system log files. On finding particular messages it will notify a human much more visibly and directly than the original log message. In this respect SWATCH is a filter/amplifier or signal cleaning tool. See Figure 1.

Reactors lead naturally to another idea: that of back-reaction, or feedback [37]. If one system can respond to a change in another, then the first system should be able to re-adapt to the changes brought about by the second system, forming a loop. For instance, cfengine can examine the state of a Unix system and run corrective algorithms. Now suppose that cfengine logs the changes it makes to the system so that the final state is known. These changes could then be re-analyzed in order to alter cfengine's program the next time around. In fact, anticipating the need for cooperative behavior, cfengine already has the necessary mechanisms to respond to analyses of the system: if its internals are insufficient, plug-in modules can be used to extend its capabilities. This interaction with modules allows cfengine to communicate with third party systems and act back on itself, adapting its program dynamically in response to changes in its perceived classification of the environment. This is essential to cfengine's converging semantics.

At the network level, the same idea could be applied to dynamical packet filtering or rejection. Network analyses based on protocols can be used to detect problem conditions and respond by changing access control lists or spam filters accordingly. The mechanisms for this are not so easily implemented today since much filtering takes place in routers which have no significant operating system, but adaptive firewalls are certainly a possibility.

Security

Security is a thorny issue. Security is about perceived threats: it is subjective and needs to be related to a security policy. This sets it on a pedestal in a general discussion on system health, so I do not want to discuss it here. Nevertheless, a healthy system is inherently more secure by any definition and security can, in principle, be dealt with in a similar manner to that discussed here [38, 39]. Since network security is very much discussed at present [40, 39, 41, 42, 43, 44], I focus only on the equally important but much more neglected issue of stability.

Biological and Social Immunity

The body's immune system deals with threats to the operation of the body using a number of pro-active and reactive systems. We can draw important lessons and inspiration from the annals of evolution. There are three distinct processes in the body: those which fight

infection, those which purge waste products and those which repair damaged tissues.

Prevention

The first line of defense against infectious disease in the body is the skin. The skin is a protective fatty layer in which most bacteria and viruses cannot survive. The skin is the body's firewall or viral gatekeeper, and with that we need not say more. The stomach and gut are also well protected by acid. Only one in ten million infectious proteins entering the body orally actually penetrate into the interior, most are blocked or broken down in the gut.

Another mechanism which prevents us from poisoning ourselves is the cleansing of waste products and unwanted substances from the blood. Natural killer cells, phagocytes and vital organs cooperate to do this job. If the blood were not cleansed regularly of unwanted garbage, it would soon be so full of cells that we would suffocate and our veins and arteries would clog up. In a similar way we can compare the performance of a system with and without a tool, like cfengine, to carry out essential garbage collection. In social systems, buildings or walls keep incompatible players apart. In computing systems one has object classes and segmentation to perform the same function. *Ultimately computer systems need to learn to distinguish illness from health, or good from bad. If such criteria can be defined in terms of computable states and policies, then illness prevention can be automated and an immune system can be built.*

Infection or Attack

When the body is infected or threatened, it mobilizes cells called lymphocytes (or B and T cells) to deal with the threat. 'Antigens', (antibody-generating threats) are often thought of as entities which are foreign to the body, but this is not necessarily the case. Complex systems are quite capable of poisoning themselves.

Normally cells in the body die by mechanisms which fall under a category known as *programmed cell death* (apoptosis). In this case, the cells remain intact but eventually cease to function and shrivel up (analogous to death signalled by the child-done signal SIGCHLD). Cells attacked by an infection die explosively, releasing their contents (analogous to a segmentation fault signal SIGSEGV), including proteins into the ambient environment. This is called *necrosis*. In one compelling theory of immunology, this unnatural death releases proteins into the environment which signal a crisis and activate an immune response. This provides us with an obvious analogy to work with.

The immune system comprises a battery of cells in almost every bodily tissue which have evolved to respond to violent cell death, both fighting the agents of their destruction and cleaning up the casualties of war: B-cells, T-cells, macrophages and dendritic cells to name but a few. Antigens are cut up and presented to T cells. This activates the T cells, priming them to

attack any antigens which they bump into. B-cells secrete antibody molecules in a soluble form. Antibodies are one of the major protective classes of molecules in our bodies. Somehow the immune system must be able to identify cells and molecules which threaten the system and distinguish them from those which *are* the system. An important feature of biological lymphocytes is the existence of receptor molecules on their surfaces. This allows them to recognize cellular objects. Recognition of antigen is based on the complementarity of molecular shapes, like a lock and key.

The canonical theory of the immune system is that lymphocytes discriminate between self and non-self [45, 46, 47, 48, [49] (part of the system or not part of the system). This theory suffers from a number of problems to do with how such a distinction can be made. Foreign elements enter our bodies all the time without provoking immune responses, for instance during eating and sex. The body has its own antigens to which the immune system does not respond. This leads one to believe that self/non-self discrimination as a human concept can only be a descriptive approximation at best; from a computer viewpoint it would certainly be a difficult criterion to program algorithmically. Recent work on the so-called *danger model* [50] proposes that detector cells notice the shrapnel of non-programmed cell death and set countermeasures in motion. A dendritic cell attached to a body cell might become activated if the cell to which it is attached dies; the nature of the signalling is not fully understood. Although controversial, this theory makes considerable sense algorithmically and suggests a useful model for computer immunity: signals are something we know how to do.

The immune system recognizes something on the order of 10^7 different types of infectious protein at any given time, although T-cells have the propensity to detect a repertoire of 10^{16} and B-cells 10^{11} [51]. Apart from being a remarkable number to contemplate, the way nature accomplishes this provides some ingenious clues as to how an artificial immune system might work. The immune response is not 100 percent efficient: it does not recognize every antigen with complete certainty. In fact it is only something on the order of 10^{-5} or 0.001 percent efficient. What makes it work so effectively, in the face of this inefficiency, is the large number of cells in circulation (on the order of 10^{12} lymphocytes). There is redundancy or parallelism in the detection mechanism. Since the cells patrolling the body for invaders rely on spot checks, it is necessary to compensate for the contingency of failure by making more checks. In other words, the body does not set up roadblocks which check every cell's credentials: it relies on frequent random checks to detect threats. Indeed, there would not be room in the body for a fighting force of cells to match every contingency so new armies must be cloned once an infection has been recognized. The dead or marked cells are

consumed by the body's garbage collection mechanism: macrophages 'eat' any object marked with an antibody. Phagocytes are the cells which engulf dead cells and remove them from the system.

Originally it was believed [52] the body was able to manufacture antibody only after having seen invading antigens in the body. However later it was shown [53, 54] that the body can make antibody for an antigen which has never existed in the history of the world. Having a repertoire with predetermined (random) shapes, the body uses a method of Darwinian (clonal) selection. Cells which are recognized proliferate at the expense of the rest of the population. The computer analogy would be to create a list of all possible checks and to change the priority of the checks in response to registered attacks. Seldom used attacks migrate down the list as others rise to the forefront of attention. This is also closely related to neural behavior and suggests that neural computing methods would be well suited to the task. Learning in a neural net is accomplished by random selection provided there is a criterion of value which selects one neural pathway over another when the correct random pathway is selected. In the case of a learning baby making random movements to grasp objects, the (presumably genetically inherited) criterion is the 'pleasure of success' in targeting the objects. In building a system of automatic immunity based on cheaply computed principles, the basic criteria for good and evil, or healthy and sick which must be determined first.

The message is this: autonomous systems do not have to be expensive provided the system holds down the number of challenges it has to meet to an acceptable level. In the body, the immune system does not maintain a huge military presence in the body at all times. Rather it has a few spies which are present to make spot checks for infection. The body clones armies as and when it needs them. Inflammation of damaged areas signals increased blood flow and activity and ensures a rapid transport of cloned killer cells to the affected area as well as a removal of waste products. In a similar way, computers could alter their level of immune activity if the system appeared abnormal. Balance through feedback is important though: cancer is one step away from cloning.

Biological protocols

Protocols, or standards of behavior, are the basic mechanism by which orderliness and communication are maintained in complex systems. In the body a variety of protocols drive the immune system. The immune system encounters intruders via a battery of elements: antibody markers, T-cell presenting cells, the Peyer's patches in the gut and so on [55]. In social systems one has rules of behavior, such as: put out the garbage on Tuesdays and Fridays; put money in the parking meter to avoid having your car swallowed by some uniformed phagocyte and so on.

Presently the main protocol for dealing with failure in the computing world is the 2001 syndrome [56]: wait for the system to collapse and then fix it. Complete collapse followed by reboot. No other organization or system in the world functions with such a singular disregard for its own welfare and the welfare of its dependents (the users). If a light bulb burns out and we replace it, there is no significant loss to its dependents. If a computer crashes users can lose valuable data, not merely time. Protocol solutions need to be built into the fabric of operating systems.

Computer Immunity and Repair

Computer Lymphocytes

What can we adapt from biological systems in order to build not merely fault tolerant systems, but fault correcting systems? Are the mechanisms of natural selection and defensive counter attack useful in computer systems?

The main difference between a computer system and the body is that the numbers are so much smaller in a computer system that the discrete nature of the system is important. Pattern recognition is a useful concept, but how should it be applied? The recognition of patterns in program code could be applied to individual binaries and might be used to detect potentially harmful operations, such as programs which try to execute "rm -rf *" or which attempt to conceal themselves using standard tricks. In order to select programs-to-allow and programs-to-reject one must search for code strings which can lead to dangerous behavior.

Self/non-self is not a very useful paradigm in computing and some immunologists believe that it is also an erroneous concept in biology. It is clearly irrelevant where a program originates; indeed we are actively interested in obtaining software from around the world. Such transplants or implants are the substance of the Internet. Rather, we could use a *danger model* [50] to try to detect programs which exhibit dangerous behavior as they run. The danger model in biology purports that the immune system responds to chemical signals which are leaked into the environment through the destruction of attacked cells. In other words, it is things which cause damage which activate an immune response. Here we shall define a danger model to be one in which an immune response is based on the general detection of dangerous conditions in the system. An immune system lies dormant until a problem is detected and wakes up in response to some signal of damage. This is the opposite of the way a firewall works, or preventative philosophies such as the security model the Java virtual machine [57]. In an immune system one already admits the defeat of prevention.

Today, the necessary danger signals might be found in the logs of programs running in user mode, or from the kernel exec itself. Ideally programs would

not just log alerts to syslog, they would be able to activate a response agent (a lymphocyte) to fight the infection, i.e., the logging mechanism would be a reactor like inetd or listen, not just a dumb receptor [23].

A more efficient danger model for the future could be constructed by introducing a new standardized signalling mechanism. If each system process had a common standard of signalling its perceived state (in addition to, and different from, the existing signals.h) this could be used to calculate a vector describing the collective state of the system. This could, in turn, be used to create advanced feedback systems, discussed in the fourth section. To diagnose the correct immune response, programs need to be able to signal their perceived state to the outside world. Normally this is only done in the event of some catastrophe or on completion, but computer programs are proportionally more valuable to a computer than cells are to the body and we are interested in the effect each program has on the totality of the system. A program is often in the best position to know what and when something is going wrong. Outside observers can only guess. In some ways this is the function of system logs today, but the information is not in a useful form because it is completely non-standard and cannot be acted upon by the kernel or an immune system. To provide the simplest picturesque example of this kind of signalling, consider the characterization of running processes by the basic 'emotional' states or the system weather:

- Happy/Sunny (Plenty of resources, medium activity)
- Sad/Cloudy (Low on resources)
- Surprised/Unsettled (System is not in the state we expect, attack in progress, danger?)
- Angry/Stormy (System is responding to an attack)

Using such insider information, an immune response could be switched on to counter system stress. In order to be effective in practice, such states need to be related to a specific resource, for example: disk requirements, CPU requirements, the number of requests waiting in event queues etc. This would allow the system to modify its resource allocation policies, or initiate countermeasures, in order to prevent dangerous situations from developing. It is tempting to think of processes which could quickly pin-point the source of their troubles and obtain a response from the immune system, but that is a difficult problem and it might prove too computationally expensive in practice. Since the system kernel is responsible for resource allocation, such a scheme would benefit from a deep level of kernel cooperation. A graded signal system would be a good measure of system state, but it needs to be tied to resource usage in a specific way. See also reference [29].

Assuming that such a signalling model were implemented, how would counter-measures be initiated? In the body there are specific immune responses

and non-specific immune responses. If we think in terms of what an existing cfengine based immune system could do to counter stressed systems there are two strategies: we could blindly start cfengine with its entire repertoire of tests and medicines to see if thrashing in the dark helps, or we could try to detect and activate only particular classes within a generic cfengine program to provide a specific response. These are also essentially the choices offered by biology.

The detection of dangerous programs by the effect they have on system resources is a 'danger model.' A self-non-self model based purely on recognition requires the identification and verification of program entities. This would be computationally inefficient. New in-coming programs would have to be analyzed with detection algorithms. Once verified a program could be marked with an encryption key signature for its authenticity to prevent the immune system from repeating its lengthy analysis. Or conversely, dangerous programs could be labelled with 'antibody' to prevent them from being used. Cfengine recognizes this kind of philosophy with its 'disable' strategy of rendering programs non-executable, but it requires them to be named in advance.

What is impressive about the biological immune system is that it recognizes antigens which the body has never even seen before. It does not have to know about a threat in order to manufacture antibody to counter it. Recognition works by jigsaw pattern-identification of cell surface molecules out of a generic library of possibilities. A similar mechanism in a computer would have to recognize the 'shapes' of unhealthy code or behavior [58, 59]. If we think of each situation as begin designated by strings of bytes, then it might be necessary to identify patterns over many hundreds of bytes in order to achieve identify a threat. A scaled approach is more useful. Code can be analyzed on the small scale of a few bytes in order to find sequences of machine instructions (analogous to dangerous DNA) which are recognizable programming blunders or methods of attack. One could also analyze on the larger scale of linker connectivity or procedural entities in order to find out the topology of a program.

To see why a single scale of patterns is not practical we can gauge an order of magnitude estimate as follows [51]. Suppose the sum of all dangerous patterns of code is S bytes and that all the patterns have the same average size. Next suppose that a single defensive spot-check has the ability to recognize a subset of the patterns in some fuzzy region ΔS , i.e., a given agent recognizes more than one pattern, but some more strongly than others and each with a certain probability. Assume the agents are made to recognize random shapes (epitopes) that are dangerous, then a large number of such recognition agents will completely cover the possible patterns. The worst case

is that in which the patterns are randomly occurring (a Poisson distribution). This is the case in biology since molecular complexes cannot process complex algorithms, they can only identify affinities. With this scenario, a single receptor or identifier would have a probability of $\frac{\Delta S}{S}$ of making an identification, and

there would be a probability $1 - \frac{\Delta S}{S}$ of not making an identification, so that a dangerous item could slip through the defenses. If we have a large number n of such pattern-detectors then the probability that we fail to make an identification can be simply written,

$$P_n = (1 - \frac{\Delta S}{S})^n \approx e^{-n \frac{\Delta S}{S}}.$$

Suppose we would like 50% of threats to be identified with n pattern fragments, then we require

$$-n \frac{\Delta S}{S} \approx -\ln P_n \approx 0.7.$$

Suppose that the totality of patterns is of the order of thousands of average sized identifier patterns, then $\frac{\Delta S}{S} \approx 0.001$ and $n \approx 7000$. This means that we would need several thousand tests per suspicious object in order to obtain a fifty percent chance of identifying it as malignant. Obviously this is a very large number, and it is derived using a standard argument for biological immune systems [51], but the estimate is too simplistic. Testing code at random places in random ways is hardly efficient, and while it might work with huge numbers in a three dimensional environment in the body, it is not likely to be a useful idea in the one-dimensional world of computer memory. Computers cannot play the numbers game with the same odds as biological systems. Even the smallest functioning immune system (in young tadpoles) consists of 10^6 lymphocytes, which is several orders of magnitude greater than any computer system.

What one lacks in numbers must therefore be made up in specificity or intelligence. The search problem is made more efficient by making identifications at many scales. Indeed, even in the body, proteins are complicated folded structures with a hierarchy of folds which exhibit a structure at several different scales. These make a lock and key fit with receptors which amount to keys with sub-keys and sub-sub-keys and so on. By breaking up a program structurally over the scale of procedure calls, loops and high level statements one stands a much greater chance of finding a pattern combination which signals danger. Optimally, one should have a compiler standard to facilitate this. The executable format of a program might reveal weaknesses. Programs which do stack long-jumping or use functions gets() and scanf() are dangerous, they suggest buffer overflows and so forth. It is possible that systems could enforce obligatory segmentation management on such programs, with library hooks such as Electric Fence [60]. Unfortunately such hooks incur large performance overheads, but this

could also be optimized if operating systems provided direct support for this.

Permanent programs should be screened for dangerous behavior once and for all, while more transitory user programs could be randomly tested. In this way we effectively distinguish between self and non-self, by adoption, for the sake of efficiency. There is no reason to go on testing system programs provided there is adequate security. In periods of low activity, the system would use its inactivity to make spot checks. The most adaptable strategy would be to leave a hook in each application or service (sendmail, ftp, cfengine) which would allow a subroutine antibody to attach itself to the program, testing the system state during the course of the program's execution. Problems would then be communicated back to the system.

Another possibility is that programs would have to obey certain structural protocols which guaranteed their safety. Graham et al have introduced the notion of adaptable binary programs [61]. This is a data format for compiled programs which allows adaptable relocation of code and analysis of binary performance without re-compilation. The ability to measure information about the performance and behavior of executable binaries has exciting possibilities for security and stability, but it also opens programs to a whole new series of viral attacks which might hook themselves into the file protocol.

The biological danger model also suggests mechanisms here. It purports that a cell which dies badly signals danger. The analogue in program execution is that programs which do not end with a SIGCHLD (normal programmed death) but with SIGABRT, SIGBUS or SIGSEGV etc are dangerous; see Figure 2. If the system kernel could collect statistics about programs which died badly, it would be possible to warn about the need to secure a replacement (transplant) for a key program or to restart essential services, or even to purge the program altogether.

Signal	Cause
SIGINT	Interrupt/break or CTRL-C
SIGTERM	Terminate signal
SIGKILL	Instant death
SIGSEGV	Segmentation (memory) violation
SIGBUS	Bus error/hardware fault
SIGABRT	Abnormal termination
SIGILL	Illegal instruction
SIGIOT	Hardware fault
SIGTRAP	Hardware fault
SIGEMT	Hardware fault
SIGCHLD	Child process exiting (apoptosis)

Figure 2: Some common signals from signals.h.

In the long run, it will be necessary to collect more long term information about the system. Biological systems do this by Darwinism, by playing the

game of huge numbers. Computers will have to be more refined than this.

More Feedback Systems and Reactors

Feedback in system administrations leads to some ipowerfuldeas. Computer systems driven by economic principles can provide us with a model of coping with excess load. The Market Net project [62] is developing technologies based on the notions of a market economy. This includes protocols and algorithms which adapt to changing resource availability. Resources, including CPU time, storage, sensors and I/O bandwidth, can be traded. When resources become scarce, prices rise (i.e., priorities wane) encouraging clients to adapt their resource usage. Such a system could come under attack through fraud. A consumer of services could make deceive the resource disseminator in an attempt to divert the system's wealth. Mechanisms must be in place to recognize this kind of fraud and respond to it to prevent exploitation of the systems. The kernel, as resource manager, needs to be aware of how many clones of a particular process or thread are active, for instance, and be able to restrict the numbers so as to preserve the integrity of the system. Fixed limits might be appropriate in some cases, but clearly the performance of the system could be optimized in some sense using a feedback mechanism to regulate activity. Biological and social systems adapt in just this way and a computer immune system should be able to adapt using a mechanism of this type.

The economy model holds some obvious truth, but the analogy is not quite the right one. It misses an important point: namely that operating system survival depends not only on the fair allocation of resources, but also on the ability to collect and clean up its waste products: the fight against entropy. Natural selection (evolution) is the mechanism which extends market philosophy to the real world. It includes not just resource sharing but also the ability to mobilize antibodies and macrophages which can actively redress imbalances in system operation.

From a physicists perspective a computer is an open system: a non-equilibrium statistical system. One can expect to learn from the field of statistical physics [63], field theory [64] and neural networks [65] as can biological studies.

Protocols

Protocol solutions are common in operating systems for a wide range of communication scenarios: there is security in formality. Protocols make the business of verifying general transactions easy. When it comes down to it, most operations can be thought of as transactions and formalized by procedural rules. The advantage of a protocol is the additional control it offers; the disadvantage is the overhead it entails. It is not difficult to dream up protocols which provide assurances that system integrity is not sacrificed by individual operations. Protocol solutions for system

well-being could likely solve problem, in principle, but the cost in terms of overhead would not be acceptable. A balance must be stricken whereby basic (atomic) system transactions are secured by efficient protocols and are supplemented by checks after the fact. Still, as computing power increases, it becomes viable (and for some desirable) to increase the level of checking during the transactions themselves. Let us mention a few areas where protocol solutions could assist computer immunity.

i) Process dispatch, services and the acceptance of executable binaries from outside. Programs could be examined, analyzed and verified before being accepted by the system for execution, as with the Java Virtual Machine. Hostile programs could be marked hostile with 'antibodies' and held inert, while safe programs could be marked safe with a public key. Spot checks on existing safe programs could be made to verify their integrity, perhaps using checksums, such as md5 checksums. ii) Object inheritance with histories: program X can only be started by a named list of other programs. This is like TCP wrappers/rsmsh but within the confines of each host. A linking format allows us to place hooks in a program to which the OS can attach test programs, a bit like a debugger. In this way, one could perform spot checks at run time from within. This also opens a new vulnerability to attack, unless one restricts hooks to the system. iii) License server technology is an example of software which will only run on a given host. Could one prevent people from sending native code programs to remote systems in this way? The Internet worm only propagated between systems with binary compatibility. iv) Can we detect when a program will do harm? One could audit system calls made by the program before running it in privileged mode. Detecting buffer overflows is one of the most important problems in present day computing. Electric fence etc. Of course this kind of computer system bureaucracy will slow down systems. v) Spamming could be handled by equipping reactors with a certain dead time, as one finds in neuronal activity. Adaptive locks [37] solve this problem. They could be used to limit the availability of critical and non-critical services in different ways. For example, after each ping transaction, the system would not respond to another ping transaction for a period of t seconds.

Each of these measures makes our instantaneous computer systems closer to sluggish biological systems, so it is important to choose carefully which services should be limited in this way.

Learning Systems

Seemingly inert molecular systems have a memory of previously fought infectious agents. This is not memory in the sense of computers but a memory in the Darwinian sense formed by the continual reappraisal of the system's sense of priorities. Computers cannot work in this way: the number of players in

computer systems is many orders of magnitude too small. What they can do however is to learn from past experience.

Time series prediction is a way of predicting future behavior based on past experience. Watching logs and process signals, we can build up a pattern of activity and use it to sense difficulty. Time series detection is well established in seismology, vulcanology and astronomical observation. The only difference here is that the data form a discrete alphabet of events rather than continuous measurements. Patterns need to be established: looking for regularly occurring problems such as lack of memory or swapping/paging (thrashing) fits, which disks become full, as well as process sequences which most often lead to difficulty. Advanced state detection can recognize symptoms before they develop into a problem. Fuzzy 'logic' and behavioral pattern recognition are natural ways to diagnose developing situations such as disk-full conditions and attacks to the system. Pattern recognition and neural networks will be useful for diagnosing external attacks on the system as well as for diagnosing cases where the system attacks itself.

Logging probes like Network Flight Recorder and Bro [25, 24] can be used to collect the information, but a proper machine analysis of the data is required. System logs also need to be analyzed: can we reduce complex log messages to strings of simple characters [26]? What is the alphabet of such messages? What is the scale of the signals? At the small scale (lots of detail) we have network protocols. At the large scale (averaged changes over long times) we have statistical entropy and load patterns other measures.

Information, Time Series and Statistical Mechanics

A multitasking computer, even a stand-alone computer, is a complex system; coupled to a network, its level of complexity increases manifold. Although scarcely reaching the level of biological or social complexity, computer networks could provide us with an ideal testing ground for many issues in those fields at the same time as being worthy of study for purely practical reasons. Complex systems have been analyzed in the context of physics and biology. The methodology is well known to experts, if not completely understood. Future computer systems will benefit from the methods for unravelling complexity as the level of distribution and cooperation increases. In many ways this harks back to Asimov's psycho-history: the ability to predict social trends based on previous behavior.

Complexity in a computer system arises both from the many processes which are running in the kernel and from the distribution of data in storage. System activity is influenced by the behavior of users. Users exert a random influence on the system leading to fluctuating levels of demand and supply for resources. Overlaid across this tapestry of fluctuating

behavior we can also expect some strong regular signals. We expect to find a number of important regularities: daily, hourly, and weekly patterns are to be expected since these are the frequencies with which the most common cron jobs are scheduled. They also correspond to the key social patterns of work and leisure amongst the users of the system. All students rush to the terminal room at lunch time to surf the web; all company employees run from the terminal room at lunch time to sit in the sun. The daily signal will perhaps be the strongest since most humans and machines have a strong daily cycle.

Home grown periodic behavior is easily dealt with: if we expect it, it does not need to be analyzed in depth. However, other periodic signals might reflect regular activity in the environment (the Internet for instance) over which we have no control. They would include everything from DNS domain transfers to programmed port scanning. They affect our own systems, in perhaps subtle but nonetheless important ways which reflect both the way in which network resources are shared between uncooperating parties and the habits of external users seeking their gratification from our network services. Periodic patterns can be discovered in a variety of ways: by Fourier analysis and by search algorithms, for instance. A further possibility which has important potential for the general problem of behavior analysis is the use of neural networks. Neural networks lead us into the general problem.

In a complex system, it is not practical to keep track of every transaction which occurs, nor is it interesting to do so. Many events which take place cause no major changes in the system; there are processes constantly taking place, but their effect on average is merely to maintain the status quo. In physics one would call this a dynamical equilibrium and random incoherent events would be called noise. Noise is not interesting, but a clear signal or change in the system average behavior is interesting. We are interested in following these major changes in computer systems since they tell us the overall change in the behavior with time; see Figure 3. On a stable system we would not expect the average behavior of the system to change very much. On an unstable system, we would expect large changes.

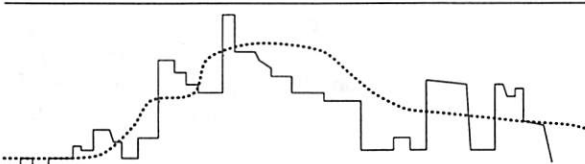


Figure 3: Although the details of system behavior seem random, the averages can reveal trends which are simpler to deal with.

The implication in the preceding sentence is based on the prejudice that significant change is a bad thing. That point of view might be criticized. What

makes the gist of the argument correct is that it is always possible to define a measured quantity in such a way that this is true. A certain level of chaos might be acceptable or even desirable, according to one definition of chaos, but unacceptable according to another. In other words, the formulation of the problem is central. The identification of the correct metrics is a subject for future research, probably more lengthy and involved than one might think.

There is a close analogy here with the physics of complex systems. At the simplest level the equilibrium state of a system and its average load has a thermodynamical analogy: namely in terms of quantities analogous to temperature, pressure and entropy. If one imagines defining a system's average temperature and pressure from the measured averages of system activity, then it is reasonable that these will follow a normal thermodynamic development over long times. From a physical point of view, a computer shares many features in common with standard thermodynamical models. The idea of using average parameters to characterize the behavior is similar to what programs such as *xload* or Sun's *perfmeter* do. There are also other ways [37] in which to record the local history of the system. To put it flippantly we are interested in computer weather forecasting. But there there is much more to be gained from the computation of averages than plotting line graphs to inform humans about the recent past. The ability to identify trends and patterns in behavior can allow a suitably trained autonomous system to take measures to prevent dangerous situations from occurring before they become so serious that it becomes necessary to fetch a 'doctor.' The reason why single messages are insufficient is that computer systems are clearly to a large extent at the mercy of users' behavior. If one understands local habits and work patterns, then preventative action can be diagnosed and administered without having to rely on the immediate availability of humans doctors and technicians. Long term patterns cannot necessarily be understood from singular log messages or threshold values of system resources. There are too many factors involved. One must instead grasp the social aspect of system usage in an approximate way.

It is interesting to remark that, by averaging over the discrete behavior of a complex system, one can end up with continuously varying potentials; see Figure 4. Possibly computer networks will at some stage of the future be reinterpreted as analogous electric circuits in which the potentials are not electricity but statistical events characterizing the flow of activity throughout. Simple conservation arguments should be enough to convince anyone that what one ends up with is simply the physics of an abstract world forged by the imprint of information flows. Much of this is implicit in Shannon's original work on information theory [66]. It should be emphasized that the physics of complex cooperative systems is one of the most difficult unsolved problems of our time so quick answers

can easily be discounted. Nonetheless, there is cause for optimism: often complexity is the result of simple transactions and simple mechanisms. My guess is that, to a useful level of approximation, the analysis of computer systems will prove to be relatively straightforward, using the physics of today, just as biological studies are benefitting from such theoretical ideas.

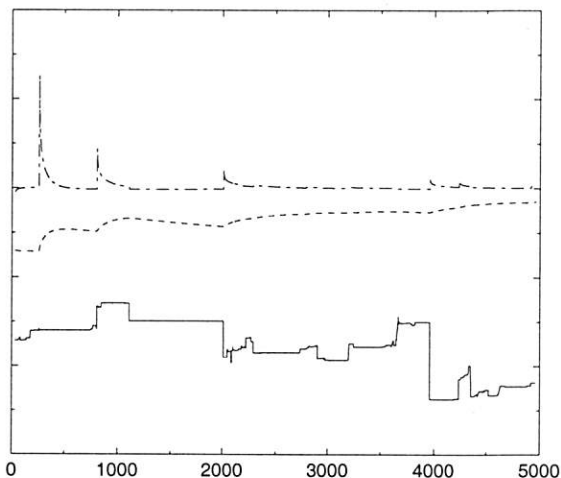


Figure 4: Disk usage as a function of time over the course of a week, beginning with Saturday. The lower solid line shows actual disk usage. The middle line shows the calculated entropy of the activity and the top line shows the entropy gradient. Since only relative magnitudes are of interest, the vertical scale has been suppressed. The relatively large spike at the start of the upper line is due mainly to initial transient effects. These even out as the number of measurements increases.

Summary: Putting the Pieces Together

All of the ideas noted in this paper have been discussed previously in unrelated academic contexts. The expertise required to build a computer immune system exists in fragmented form. What is now required is a measure of imagination and a considerable amount of experimentation in order to identify useful mechanisms put together the pieces into a

working model. Fortunately there is no shortage of ingenuity and willingness to participate in this kind of experimentation in the system administration community.

The best immune system one could build today would be made up the elements such as those in Table 1.

With these tools, each host is as self-contained as possible, accepting as little outside data as can be. Sharing of Bro/Network Flight Recorder data should be done carefully to avoid it being used as a means of manipulating the system. In the absence of a better running analysis, it is difficult to do better than this. Even so, with carefully thought out rules, this provisional approach can be very successful. Unfortunately, finding the best rules is presently a time-consuming job for an experienced system administrator. In time, perhaps we shall assemble a generic database of rules for cfengine and related tools.

Hopefully a computer immune system will at some time in the future become a standard. The last thing we need is a multitude of incompatible systems from a multitude of vendors. Free software such as GNU/Linux could blaze this trail since it is open for development and modification in all its aspects and could prevent important mechanisms from being patented. Few vendors are quick to adopt new technology, but one might hope that a properly designed fault preventive system would be more than they could resist. A POSIX standard which laid the groundwork for computer immunology is something to aim for. Future papers on this subject must lay down the operating system requirements for this to happen.

Am I trying to send the message that system administration is a pointless career, an inferior pursuit? No, of course not. An immune system cannot no more replace the system administrator than a lymphocyte can replace a surgeon, but an immune system makes the surgeon's existence bearable, fighting the stuff that is not easy to see and requiring basically no intelligence. Many of the ideas in this paper have an artificial intelligence flavor to them, but the main point is that immune systems in nature are far from

Convergence	cfengine	<i>Build expert systems for configuration Correlate system state and activity using switches or 'classes' to activate responses.</i>
Detection	Bro/N.F.R.	<i>Intrusion detection based on event occurrence. simple analysis switches on classes in cfengine with predetermined rules to counter intrusion attempts.</i>
	load average	<i>Process load average used to detect thresholds which switch feed back class data to cfengine. Cfengine restricts access to services, kills offending processes etc.</i>

Table 1: A makeshift immune system today. The key points this addresses are convergence and adaptive behavior.

intelligent. The less intelligent our autonomic systems are the cheaper they will be. Nature shows us that responsive system's don't need much intelligence as long as their mechanisms are ingenious! Simplicity and frequency are the keywords. I hope that the next few years will see important advances in the development of cooperative systems with the task of preserving the general health and reliability of the network.

I am grateful to Ketil Danielsen for a discussion about market economies in computing.

Note Added

After completing this paper, I was made aware of reference [67] where the authors conduct a time-series analysis of Unix systems very similar to those which I have advocated here. This paper deserves much more attention than I have been able to give it before the submission deadline.

Author Information

Mark Burgess is associate professor of physics and computer science at Oslo College. He is the author of GNU cfengine and can be reached at <http://www.iu.hioslo.no/~mark>, where you will also find all the relevant information about cfengine and computer immunology.

Bibliography

- [1] John Brunner. *The Shockwave Rider*. Del Rey, New York, 1975.
- [2] M. W. Eichin and J. A. Rochlis. "With microscope and tweezer: an analysis of the internet worm." *Proceedings of 1989 IEEE Computer Society symposium on security and privacy*, page 326, 1989.
- [3] M. Burgess. "A site configuration engine." *Computing systems*, 8:309, 1995.
- [4] M. Burgess and R. Ralston. "Distributed resource administration using cfengine." *Software practice and experience*, 27:1083, 1997.
- [5] M. Burgess. "Cfengine as a component of computer immune-systems." *Norsk informatikk konferanse*, page (submitted), 1998.
- [6] M. Burgess and D. Skipitaris. "Acl management using gnu cfengine." *USENIX ;login:*, Vol.23 No. 3, 1998.
- [7] Isaac Asimov. *I, Robot*. 1950.
- [8] P. Anderson. "Towards a high level machine configuration system." *Proceedings of the 8th Systems Administration conference (LISA)*, 1994.
- [9] M. Fisk. "Automating the administration of heterogeneous LANs." "Proceedings of the 10th Systems Administration conference (LISA)," 1996.
- [10] J. P. Rouillard and R. B. Martin. "Config: a mechanism for installing and tracking system configurations." *Proceedings of the 8th Systems Administration conference (LISA)*, 1994.
- [11] J. Finke. "Automation of site configuration management." *Proceedings of the 11th Systems Administration conference (LISA)*, page 155, 1997.
- [12] M. Burgess and D. Skipitaris. "Adaptive locks for frequently scheduled tasks with unpredictable runtimes." *Proceedings of the 11th Systems Administration conference (LISA)*, page 113, 1997.
- [13] R. Evard. "An analysis of unix system configuration." *Proceedings of the 11th Systems Administration conference (LISA)*, page 179, 1997.
- [14] D. Schmidt. "Ace. adaptive communication environment." <http://http://siesta.cs.wustl.edu/schmidt/ACE.html>.
- [15] Tivoli systems/IBM. *Tivoli software products*. <http://www.tivoli.com>.
- [16] Hewlett Packard. *Openview*.
- [17] Sun Microsystems. *Solstice system documentation*. <http://www.sun.com>.
- [18] Host factory. *Software system*. <http://www.wv.com>.
- [19] W. F. Tichy. "RCS - A system for version control." *Software practice and experience*, 15:637, 1985.
- [20] Caldera. *COAS project*. <http://www.caldera.com>.
- [21] *Webmin project*. <http://www.webmin.com>.
- [22] Palantir. The palantir was a project run by the university of Oslo Centre for Information Technology (USIT). Details can be obtained from palantir@usit.uio.no and <http://www.palantir.uio.no>. I am informed that this project is now terminated.
- [23] S. E. Hansen and E. T. "Atkins. Automated system monitoring and notification with Swatch." *Proceedings of the 7th Systems Administration conference (LISA)*, 1993.
- [24] V. Paxson. "Bro: A system for detecting network intruders in real time." *Proceedings of the 7th USENIX security symposium*, 1998.
- [25] M. J. Ranum, et al. "Implementing a generalized tool for network monitoring." *Proceedings of the 11th Systems Administration conference (LISA)*, page 1, 1997.
- [26] R. Emmaus, T. V. Erlandsen, and G. J. Kristiansen. *Network log analysis*. Oslo College dissertation., Oslo, 1998.
- [27] S. Kittur, et al. "Fault tolerance in a distributed chorus/mix system." *Proceedings of the USENIX Technical conference*, page 219, 1996.
- [28] W. Rosenbery, D. Kenney, and G. Fisher. *Understanding DCE*. O'Reilly and Assoc., California, 1992.
- [29] J. S. Plank. "A tutorial on Reed-solomon coding for fault tolerance in RAID-like systems." 27:995, 1997.

- [30] R. Pike, D. Presotto, S. Dorwood, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom. "Plan 9 from Bell Labs." *Computing systems*, 8:221, 1995.
- [31] S. J. Mullender, G. Van Rossum, A. S. Tannenbaum, R. Van Renesse, and H. Van Staveren. "Amoeba: a distributed operating system for the 1990s." *IEEE Computer*, 23:44, 1990.
- [32] A. S. Tannenbaum, R. Van Renesse, H. Van Staveren, G. J. Sharp, S. J. Mullender, J. Jansen, and G. Van Rossum. "Experiences with the amoeba distributed operating system." *Communications of the ACM*, 33:46, 1990.
- [33] G. D. Parrington, S. K. Shrivastava, S. M. Wheeler, and M. C. Little. "The design and implementation of arjuna." *Computing systems*, 8:255, 1995.
- [34] The Object Management Group. "Corba 2.0, interoperability: Universal networked objects." *OMG TC Document 95-3-10*, Framingham, MA, March 20, 1995.
- [35] H. Wasserman and M. Blum. "Software reliability via run-time result-checking." *J. ACM*, 44:826, 1997.
- [36] Mark Burgess. *GNU cfengine*. Free Software Foundation, Boston, Massachusetts, 1994-1998.
- [37] M. Burgess. "Automated system administration with feedback regulation." *Software practice and experience*, (To appear), 1998.
- [38] M. Carney and B. Loe. "A comparison of methods for implementing adaptive security policies." *Proceedings of the 7th USENIX security conference*.
- [39] N. Minsky and V. Ungureanu. "Unified support for heterogeneous security policies in distributed systems." *Proceedings of the 7th USENIX security conference*.
- [40] SANS. "System administration and network security." <http://www.sans.org>.
- [41] USENIX. "Operating system protection for fine-grained programs." *Proceedings of the 7th USENIX Security symposium*, page 143, 1998.
- [42] I. S. Winkler and B. Dealy. "A case study in social engineering." *Proceedings of the 5th USENIX security symposium*, page 1, 1995.
- [43] J. Su and J. D. Tygar. "Building blocks for atomicity in electronic commerce." *Proceedings of the 6th USENIX security symposium*:97, 1996.
- [44] W. Venema. "Murphy's law and computer security." *Proceedings of the 6th USENIX security symposium*, page 187, 1996.
- [45] F. M. Burnet. *The Clonal selection theory of acquired immunity*. Vanderbilt Univ. Press, Nashville TN, 1959.
- [46] F. M. Burnet and F. Fenner. *The production of antibodies*. Macmillan, Melbourne/London, 1949.
- [47] J. Lederberg. *Science*, 1649:129, 1959.
- [48] R. E. Billingham, L. Brent, and P. B. "Medawar." *Nature*, 173:603, 1953.
- [49] R. E. Billingham. *Proc. Roy. Soc. London.*, B173:44, 1956.
- [50] P. Matzinger. "Tolerance, danger and the extended family." *Annu. Rev. Immun.*, 12:991, 1994.
- [51] A. S. Perelson and G. Weisbuch. "Immunology for physicists." *Reviews of Modern Physics*, 69:1219, 1997.
- [52] Linus Pauling and Dan H. Campbell. "The production of antibodies in vitro." *Science*, 95:440, 1942.
- [53] G. Edelman. (unknown reference), later awarded Nobel prize for this work in 1972 with R. R. Porter, 1959.
- [54] R. R. Porter. (unknown reference), later awarded Nobel prize for this work in 1972 with G. Edelman, 1959.
- [55] I. Roitt. *Essential Immunology*. Blackwell Science, Oxford, 1997.
- [56] A. C. Clarke and S. Kubrick. *2001: A space odyssey*. MGM, Polaris productions, 1968.
- [57] I. Goldberg, et al. "A secure environment for untrusted helper applications." *Proceedings of the 6th USENIX security symposium.*, page 1, 1996.
- [58] Sun Microsystems. *Java programming language*. <http://java.sun.com/aboutJava/>.
- [59] C. Cowan, et al. *Stackguard project*. <http://www.cse.ogi.edu/DISC/projects/immunix/StackGuard/>.
- [60] Pixar, B. Perens. "Electric fence, malloc debugger." *Free software foundation*, 1995.
- [61] S. Graham, S. Lucco, and R. Wahbe. "Adaptable binary programs." *Proceedings of the USENIX Technical conference.*, page 315, 1995.
- [62] Market net project. *A survivable, market-based architecture for large-scale information systems*. <http://www.cs.columbia.edu/dcc/MarketNet/>.
- [63] F. Reif. *Fundamentals of statistical mechanics*. McGraw-Hill, Singapore, 1965.
- [64] Mark Burgess. *Applied covariant field theory*. <http://www.iu.hioslo.no/mark/physics/CFT.html>, (book in preparation).
- [65] J. A. Freeman and D. M. Skapura. *Neural networks: algorithms, applications and programming techniques*. Addison Wesley, Reading, 1991.
- [66] C. E. Shannon and W. Weaver. *The mathematical theory of communication*. University of Illinois Press, Urbana, 1949.
- [67] P. Hoogenboom and J. Lepreau. "Computer system performance problem detection using time series models." *Proceedings of the USENIX Technical Conference, Summer 1993*, page 15, 1993.

A Visual Approach for Monitoring Logs

Luc Girardin and Dominique Brodbeck – UBS, Ubilab

ABSTRACT

Analyzing and monitoring logs that portray system, user, and network activity is essential to meet the requirements of high security and optimal resource availability. While most systems now possess satisfactory logging facilities, the tools to monitor and interpret such event logs are still in their infancy.

This paper describes an approach to relieve system and network administrators from manually scanning sequences of log entries. An experimental system based on unsupervised neural networks and spring layouts to automatically classify events contained in logs is explained, and the use of complementary information visualization techniques to visually present and interactively analyze the results is then discussed.

The system we present can be used to analyze past activity as well as to monitor real-time events. We illustrate the system's use for event logs generated by a firewall, however it can be easily coupled to any source of sequential and structured event logs.

Introduction

One of the primary sources of information that enable support for system administration is the logging facilities provided by key system components. Such facilities are often used to track real-time events triggered by system, user, and network activity. Continuously monitoring this activity is of tremendous importance to organizations which rely on high security [Geer, et al., 1997] and optimal resources availability. Logs provide support for proactive system maintenance, anomaly and intrusion detection, failure analysis, and usage assessment.

Conscientious system and network administrators often scan entire log files on a regular basis with little or no tool support. However, the growing use of interconnected computer systems and of the Internet has resulted in a drastic increase of the amount of information contained in such logs, making sequential scans impractical. Existing tools typically utilize rule-based systems to eliminate known-good log entries, and statistical abstractions to simplify the analysis of the activity. Such tools greatly relieve from the repetitive work of scanning log files but generate new difficulties.

The rule creation process forces administrators to draw a clear distinction between what is relevant/irrelevant, severe/benign, critical/isolated, etc... Furthermore, capturing and translating the tacit knowledge of the domain expert into a formal set of rules is difficult and prone to errors. In addition, the resulting rule set usually only covers well-known behaviors, while the most interesting events are often a result of unforeseeable behaviors. Finally, achieving a smooth evolution of the base rule set remains more an art than a science.

The statistical approach, also referred to as data mining, is good at providing abstraction but usually ignores the complexity and the context in which the

activity takes place. Moreover, it doesn't support exploratory tasks, such as finding the reason for an increasing number of a certain type of network packets.

Information visualization techniques take advantage of the human perceptual capabilities and provide an overview of the global relationships within a data set while still providing for detailed examination of the underlying data. Examples of visualization techniques for the monitoring of logs may be found in [Couch, et al., 1996, Hughes, 1996, Karam, 1994].

With machine learning, the goal is to provide automatic classification of the events to permit unattended operations. This approach does usually not rely on any prior knowledge about the content of the data. Applications focussing mainly on computer security, can be found in [Hofmeyr, et al., 1998, Lankewicz, et al., 1997, Tan, 1997].

While both the information visualization and machine learning approaches are promising, no tool combines the potential of both for monitoring event logs. By taking advantage of their strengths, it is possible to provide humans with an interactive display to better understand the activity taking place, while delegating the repetitive tasks to the computer.

Following, we discuss our set of tools, which use a variant of the self-organizing map algorithm and a spring-based layout engine to act as a classifier through multidimensional scaling and topological clustering. To visualize and interact with these representations we make use of the map metaphor, and complementary information visualization techniques such as parallel coordinates and dynamic range sliders to facilitate their investigation.

Visually exploring the log activity provides us with new ways to analyze what is going on and to discover hidden patterns. Our set of components provides

for more effective and user-friendly monitoring of event logs since it focuses on the complexity, letting the computer take care of the simple tasks. Our tools specifically addresses the problems of real-time monitoring and heterogeneous event attributes.

Overview

A logging facility usually stores the activity as a time ordered sequence of events. Logging facilities provide either structured or unstructured descriptions of events. Structured logs contain distinct and uniform entries characterizing each event by a given set of attributes. Most systems, such as firewalls (see Figure 1), accounting systems, and web servers, usually provide us with such structured information.

Unstructured logs do not necessarily contain uniform event entries. For example, the popular syslog facility stores free text entries associated with each event. The individual attributes within such a string may be identified using an indexing technique similarly to the one described in [Chen, et al., 1998] or by judicious parsing.

The attributes form an n -dimensional space, where n is the number of attributes. Each event has a unique position in this space depending on the specific values of its attributes. To put each event into context, we must be able to compare one event to another using a distance or similarity function. Such a function measures the extent to which two elements are related or similar to each other. In our experimental system, we use a modified Euclidean metric. The distance between two characteristics is naively calculated using a simple subtraction for continuous attributes such as time stamps and lengths, and a string comparison for categorical attributes where a match results in a zero, otherwise a one. The overall distance is then obtained by summing all the resulting values. This is subject to a more subjective and empirical definition when domain knowledge is available. For more detail on our metric, please refer to [Brodbeck, et al., 1997].

To provide an overview of this space and portray the activity as a whole, we need to reduce the

dimensionality in order to generate a representation of the relative similarities of events, which may be displayed on the screen. The resulting visualization can then serve as a frame of reference to drive further analysis and to embed fine grained tasks such as searching and querying. We choose our low-dimensional space to be a plane, for usability reasons.

Events which are similar will be placed close together while events with unrelated patterns will be further apart. The algorithms we use produce a mapping for each event so that their resulting locations will approximate their similarities and thus form a spatial classification. The resulting representation can be viewed as a map with the particularity of communicating abstract relationships.

We provide two information visualization techniques that support the analysis, filtering, and querying of the information viewed on the map. Parallel coordinates permit the comparison of log entries through visual presentation of their similar and dissimilar properties. Dynamic range sliders are used to highlight a subset of the log entries by interactively specifying the range of each attribute.

Multidimensional Scaling

We use two competing methods to achieve the dimensionality reduction process. Each of the two differs in their output and in their computational complexity. The first uses a method inspired from physics, the spring layout algorithm, while the other is based on an artificial neural network, the self-organizing map algorithm. These two algorithms do not depend on any prior knowledge of the data and in this sense we can say that they exhibit self-organization.

Spring Layout

The spring layout algorithm is based on a physical model where all the data points are mutually connected by springs. The rest distances of these springs are proportional to the respective similarity of the data points in high-dimensional space. The more similar two points are, the shorter the spring between them.

No	Date	Time	Inter.	Origin	Type	Action	Service	Source	Destination	Proto.	Rule	S_Port
2	20Jun98	4:30:01	hme1	pilatus	log	accept	smtp	pilatus-dmz	rigi	tcp	2	42833
3	20Jun98	4:30:01	hme1	pilatus	log	accept	domain-udp	rigi	eiger	udp	1	58942
4	20Jun98	4:30:01	hme1	pilatus	log	accept	domain-udp	rigi	eiger	udp	1	58943
5	20Jun98	4:30:01	hme1	pilatus	log	accept	domain-udp	rigi	eiger	udp	1	58944
6	20Jun98	4:30:01	hme0	pilatus	log	accept	domain-udp	eiger	d.root-servers.net	udp		
7	20Jun98	4:30:01	hme0	pilatus	log	accept	ntp-udp	anapurna	err.ee.ethz.ch	udp		
8	20Jun98	4:30:03	hme1	pilatus	log	accept	domain-udp	rigi	eiger	udp	1	58945
9	20Jun98	4:30:03	hme1	pilatus	log	accept	domain-udp	rigi	eiger	udp	1	58946
10	20Jun98	4:30:03	hme1	pilatus	log	accept	smtp	rigi	eiger	tcp	4	35713
11	20Jun98	4:30:04	hme0	pilatus	log	accept	domain-udp	eiger	ns1.sunrise.ch	udp		
12	20Jun98	4:30:06	hme0	pilatus	log	accept	dhcpd-udp	durham	102.153.89.255	udp		

Figure 1: The logging information provided by a mainstream firewall. Each event is characterized using a time stamp, the source and destination hosts, the protocol used, whether the connection has been accepted, dropped or rejected, and other relevant information.

The algorithm starts with a random arrangement of the data points in a low-dimensional space. The system is then set free and left to relax with the effect that distant data points which are similar in high-dimensional space are pulled together and close but dissimilar ones are pushed apart. After a number of iterations the system typically stabilizes, resulting in a layout of the data in a low-dimensional space where strongly correlated dimensions are blended together, and topology of the data in high-dimensional space is preserved as best as possible.

The low-dimensional space is defined as three-dimensional at the beginning of the relaxation process and the data points are randomly positioned inside a sphere. We do however introduce a gravity force which slowly pulls the data points towards the ground plane so that we eventually end up with a flat, two-dimensional layout. This process reduces the probability of data points being trapped in a local minimum.

This algorithm may be performed in linear iteration time [Chalmers, 1996]. However, it is still not suitable for datasets containing more than a few thousand entries. For significantly larger datasets, sampling, or pre-clustering using for example the self-organizing map algorithm, can provide a reasonable solution. Parallel computers can also effectively

achieve any required scalability by computing the spring forces and the positions concurrently.

The spring layout out algorithm can support real-time processing of dynamic feeds; data points can be gradually inserted at random positions, and older ones removed so as to keep the computational load constant. By shaking the system and letting it run for few iterations, it will quickly stabilize to an appropriate configuration.

Self-organizing Map Algorithm and Categorical Data

The self-organizing map algorithm [Kohonen, 1995] is inspired from biology and imitates two-dimensional maps of the brain made from sensory modalities such as the behavior of the tonotopic map of the auditory region. It is an unsupervised (self-organizing) neural network which generates a feature map that has preserved the topology of the stimuli according to their similarity. The original self-organizing map algorithm is not able to learn categorical values and a variant of the original algorithm has been developed to tackle this limitation.

Self-organizing maps are often used to decrease the number of dimensions while preserving the topological features of a high-dimensional space [Kaski,

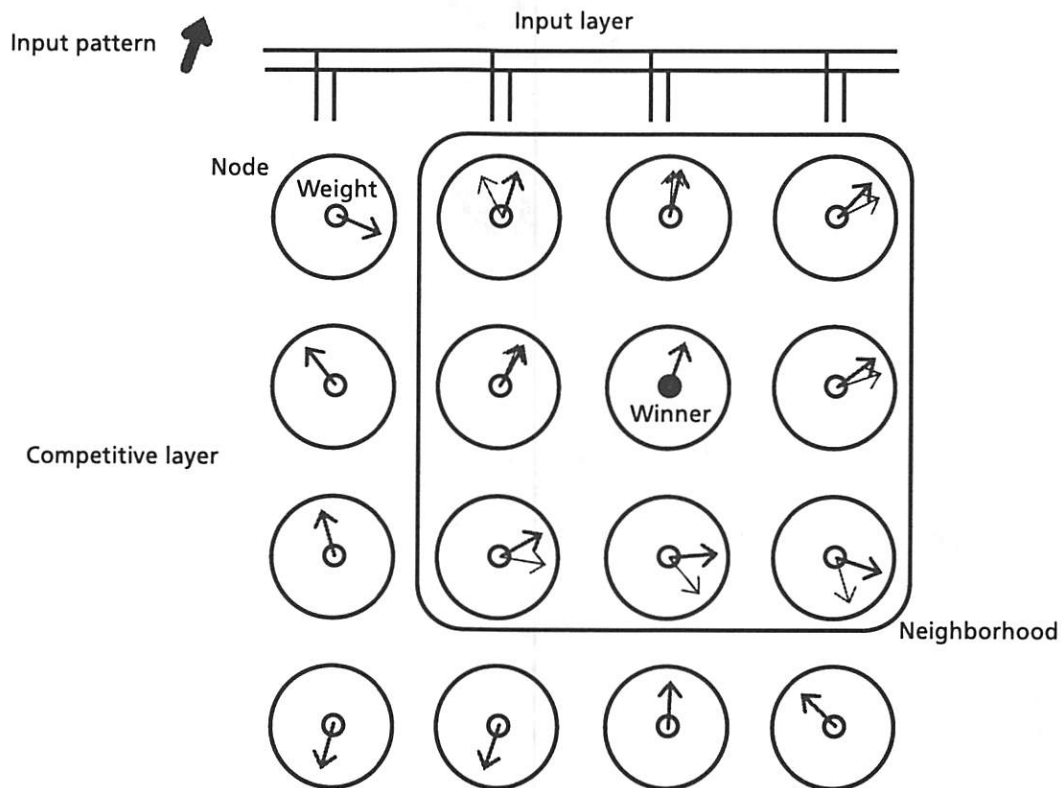


Figure 2: The adaptation process in the self-organizing map algorithm is initiated by feeding the input layer with patterns. Then, the node with a weight that match best the input pattern, the winning node, is sought in the competitive layer. Weights in the neighborhood of the winner can now be slightly adjusted to resemble more closely the input pattern.

1997]. By presenting the feature vectors (in our case the coordinates of each event as a point in an n -dimensional space) to the input layer, the neural network will adapt the weights in the competitive layer to produce a map in the output layer that will exhibit as best as possible the topology of the input space. Please refer to Figure 2 for a description of the adaption process. The nodes in the competitive layer represent a generalization of the possible input patterns.

Our variant algorithm makes use of dynamically growing dictionaries for each weight associated with a categorical dimension. The dictionary contains a set of weighted categories and new categories are dynamically added as needed. Each weight is a vector in a subspace containing only the categories relevant to its associated node. Therefore, the network can learn categorical attributes without exhausting the computing and storage capabilities of the machine.

The self-organizing map algorithm outperforms other methods [Li, et al., 1995], the most popular technique being principal component analysis (PCA). However, as the major difference between self-organizing maps and metric multidimensional scaling

techniques (such as the spring layout algorithm), the self-organizing map algorithm will only try to preserve the order of the distances in the high-dimensional space, and will not perform the more natural metric transformation.

Computing self-organizing maps of static datasets is achieved by randomly training the network with the event log entries, until convergence. The time needed to achieve convergence depends on the complexity of the input space, in contrast to the spring layout algorithm which is constrained by the number of event log entries.

Training the network can be a computing intensive process. The computational complexity can be reduced by lowering the dimensionality of the event logs (reducing the number of attributes) or by reducing the size of the output map (resulting in coarser granularity in the clustering). Alternatively, the algorithm can be easily parallelized at the node level.

To analyze logs in real-time, the network can position new event patterns on the output map. However, the results may become of poor quality if the new patterns do not follow the distribution of the

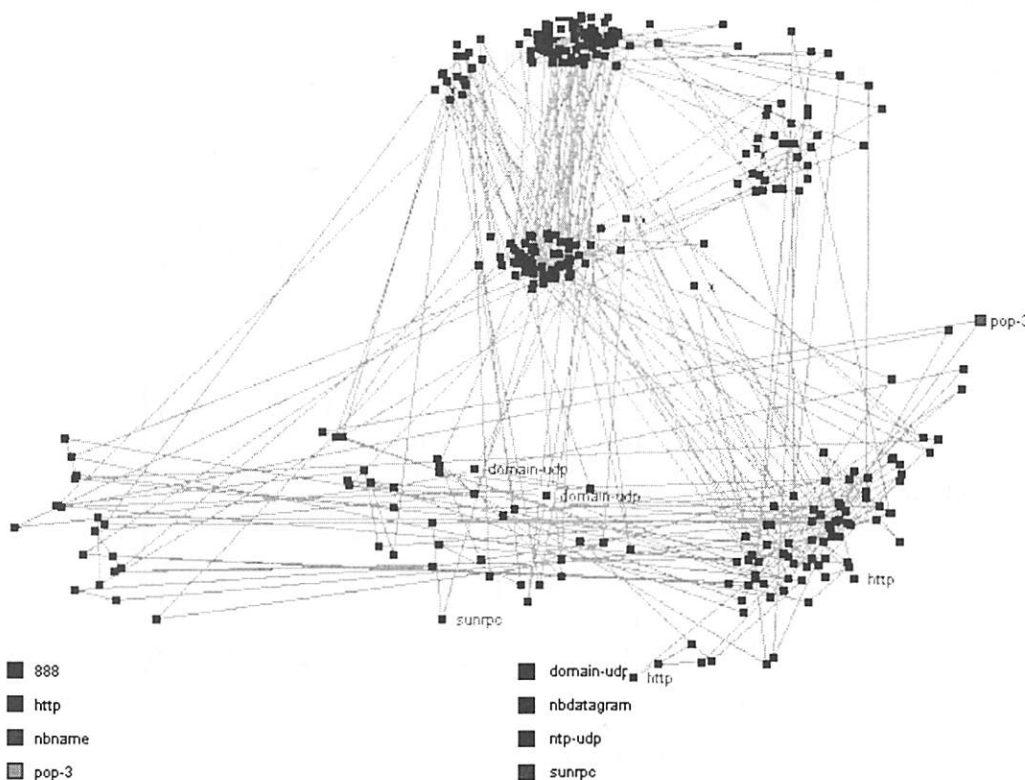


Figure 3: A spring layout of about 500 events from our firewall. Each data point is colored according to the protocol attribute. Data points are sequentially connected by times, displaying sequences of events with similar characteristics. This visualization clearly establish the similarities and relationships. We can see for example that http requests (in the lower right corner) strongly rely on the domain name service (in the center of the lower part).

events used during the training period. In such a situation, the network may need to be retrained. To avoid the retraining problem, we constantly train the network with new input patterns, and use intermittent noise generation in the weights to insure convergence. However, we are still investigating the theoretical foundation of this strategy.

Visualization

Display of the Spring Layout

To graphically present spring layouts, we use a map metaphor. This provides an overview and establishes a reference system in order to avoid 'getting lost' during the exploration. All logged events are depicted on the map and permit the visual perception of clusters and outliers. Concretely, areas with high density of data points reveal large number of similar activity patterns, while isolated data points correspond to unfamiliar events (see Figure 3). To aid the analysis of the map, some techniques have been used to increase imageability [Brodbeck, et al., 1997]. This approach is ideal for relatively small datasets for which more detailed exploration of the information is desirable. Moreover, maps portraying the activity in real-time have not yet been investigated using this

technique and may create usability problems. We currently favor it for analysis of the activity during a limited time frame and for detailed exploration of some nodes within the maps created by the self-organizing map algorithm.

Display of the Self-organizing Map

The self-organizing map results in a discrete space where the units have been topographically ordered. Therefore its display is composed of a fixed number of cells that contain similar events. We developed two visualization techniques to help comprehend the information contained in the network.

The first is to visualize for each cell the characteristics of the last event that it has been assigned. By merging colors, shapes, and textures similarly to the approach of [Levkowitz, 1997], it is possible to code multiple attributes into a single integrated iconic representation (see Figures 4-5). This permits an effective real-time visualization of the changes occurring in the system.

Alternatively, it is possible to use the information contained in the set of weights, which can be seen as a characterization of each node's most typical event. For our purpose, we extract the category with the

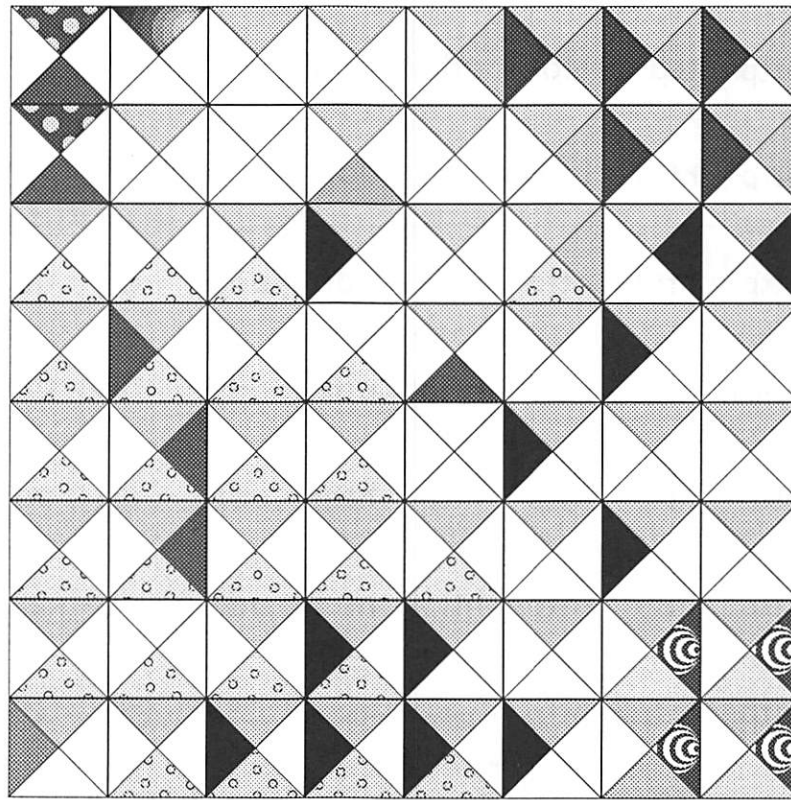


Figure 4: On this display, about 30,000 events have been clustered on a 8 by 8 grid. Each cell depicts some characteristics of the last event which was placed in this location. Four characteristics, which in this case are all categorical attributes, are depicted with triangles using a combination of patterns and colors. The main purpose of this is to provide a visual comparison of cells to determine if they are of similar nature.

dominant weight, which is then compared to the other weights to find the prevalent means of describing a cell. You can find an example in Figure 6. This process is greatly simplified when dealing with numerical values, since the characterization is the weight value itself, and such values may be represented on a continuous scale, for example using simple graphs.

Compared to the spring layout-based maps, the self-organizing map provides for more distinctive identification of clusters and is therefore well suited for larger datasets or real-time monitoring.

Parallel Coordinates

The method of parallel coordinates [Inselberg, 1985] allows the visualization of multidimensional data by a simple two dimensional representation. Through a single view, divergence, trends, and correlations can be analyzed among multiple data points or sets of data points (see Figure 7).

The parallel coordinates system is created by representing each dimension of an n-dimensional space as a vertical axis, spaced apart at a constant distance. A n-dimensional point is then constructed by

Interface	Type	Service	Source	Destination	Proto	Rule
>hme0	accept	nbname	wallace ubilab.ch	192.153.89.255	udp	17
>hme0	accept	nbname	groenland ubilab.ch	192.153.89.255	udp	17
>hme0	accept	nbdatagram	mururoa ubilab.ch	192.153.89.255	udp	17
>hme0	accept	nbdatagram	alice ubilab.ch	192.153.89.255	udp	17
>hme0	accept	nbdatagram	java ubilab.ch	192.153.89.255	udp	17
>hme0	accept	nbdatagram	mtblanc	192.153.89.255	udp	17
>hme0	accept	nbdatagram	java ubilab.ch	192.153.89.255	udp	17
>hme0	accept	nbname	durban	192.153.89.255	udp	17

Figure 5: By selecting a cell in the representation of the self-organizing map, it is possible get the details about each event it contains. In the above example, the cell contains NetBIOS announcements broadcasted on our local network.

tcp	udp	udp	udp	udp	domain-udp	domain-udp	eiger
http	tcp	tcp	udp	udp	domain-udp	domain-udp	domain-udp
http	http	http	tcp	udp	udp	domain-udp	domain-udp
http	http	http	http	udp	udp	udp	udp
america	http	http	http	udp	udp	udp	udp
http	http	http	http	http	udp	udp	udp
http	http	http	http	http	udp	udp	.255
http	http	http	mtblanc	mtblanc	.255	.255	.255

Figure 6: A view of the dominant characteristic of each cell. These values are obtained by scaling each weight over the sum of all weights divided by the number of categories. The present example shows that most of the traffic originates from http requests and domain name queries. Traffic to or from the host 'mtblanc' is mainly composed of http requests and broadcasts, while the host 'america' seems to be exclusively concerned with http requests. The host 'eiger' seems central to domain name queries; it is in fact our primary DNS server.

connection all of its attributes values on their respective axes by a polygonal line.

Dynamic Queries

Dynamic queries [Ahlberg, et al., 1995] are a means to filter event log entries. This technique works by providing a slider for each variable. When one slider is changed, the relevant points are simultaneously updated on the map. Points that fall outside of a slider's range are visually deemphasize on the map using a ghosting effect. One slider is set up for each attribute so that ranges of selected values are AND'd together to form a compound query (see Figure 8).

Practical Use

To investigate the activity taking place in the logs, we can interconnect our visual components to provide for interactive exploration. This allows the users address their multiple needs, using the appropriate tool for a particular task and taking advantage of their possible synergies. In brief, the self-organizing map is used as the global frame of reference, with spring layout maps used for more accurate depiction of interesting areas. Items can be selected on both kinds of map and visually compared using parallel coordinates. In addition, the dynamic range sliders are provided for filtering and querying, using a ghosting effect for all points that don't satisfy the conditions,

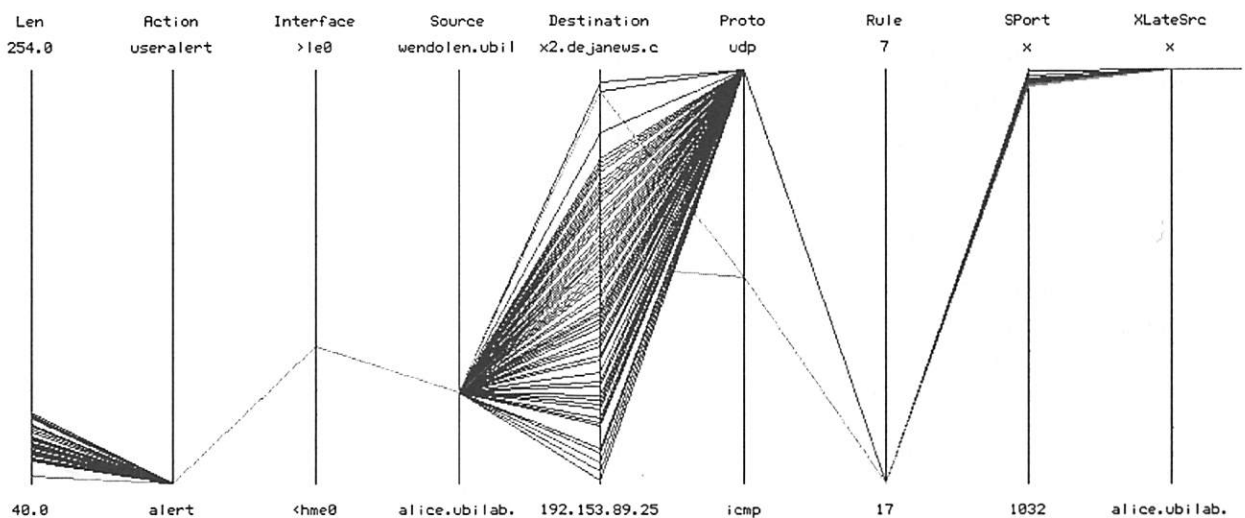


Figure 7: Parallel coordinates visualize multidimensional data points as polygonal lines. In the above example, we compare a cluster with two outliers. We see that they differ in two attributes: they are of shorter length and use the tcp protocol instead of udp.

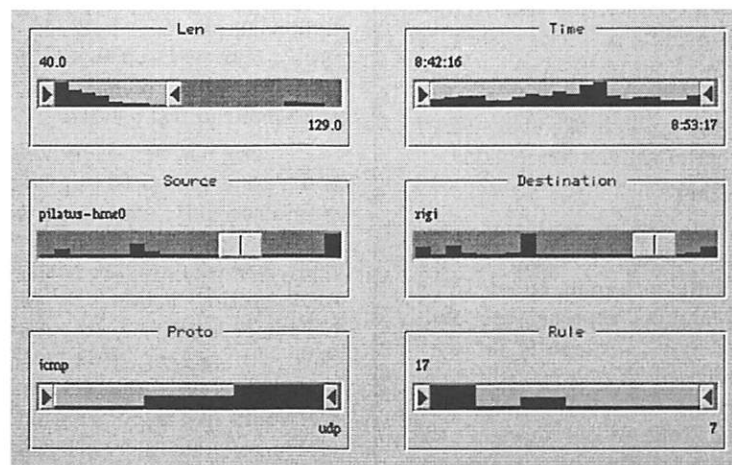


Figure 8: Dynamic queries allow for range selection over each attribute interactively. This simplifies the creation of complex queries. The query above results in the selection of connections of firewall events with short packet lengths, originating from the host 'pilatus-hme0' and having 'rigi' as the destination. Histograms are integrated into the sliders to show the distribution of data values.

effectively highlighting the rest without losing context.

By using these components to analyze the connections going through our firewall, we found some interesting applications for our approach. Below we describe a few of the more important ones, and explain how we currently use our tools to perform them.

Usage Assessment

Assessing how the resources are used and how the information flows through the network is important in respect to maximizing the use of the computing and networking infrastructures. During the analyses of our firewall logs, we discovered that two of our master DNS servers were obtaining exactly the same information, inefficiently querying remote sources for information already available locally. This information was obtained by analyzing a few adjacent cells in the self-organizing map using the spring layout. On the resulting map, some events have been laid out in the middle of two clusters containing the domain name activity of our servers. By visualizing them using parallel coordinates, it was obvious that the destination host was similar for both servers. We were able to easily correct this inefficiency and gain more network bandwidth.

Real-time Trend Analyses

By visualizing the evolution of the activity over time, one can create a mental image of the changes occurring in the system. This can be achieved by sporadically getting an overview of the present situation using the self-organizing map, or possibly using a spring layout. This way, we were able to gain a general idea of when people were mainly browsing the web, remote colleagues were sending email to us, and hackers were scanning our network.

Anomaly Detection

It is better to detect and proactively repair problems to maintain a good quality of service, than to wait for users to complain about a service behaving abnormally. Using this tool kit we were visually warned that a network segment was broken because of a high and repetitive number of similar connections.

Break-in Attempts Detection

Intrusion detection deals with the problem of distinguishing between misuse and normal use. It is clear that all types of intrusive behavior cannot be identified in advance [Frank, 1994], and our approach acknowledges this fact. While we didn't manage to discover any real intruder, we have been able to spot different sources scanning our network for security holes, and repetitive trials of abusing some of our services. We expect our approach to be more effective for intrusion detection if used with much more detailed logging information.

Discussion and Future Directions

While developing and continuously evaluating our approach, we were confronted with a number of issues, of which some remain unsolved. Since we want to remain general in our quest to make a better use of logs, we have to date ignored some opportunities to improve the effectiveness of our tools in some specific contexts, especially when domain knowledge is available.

One of the main difficulties we have been faced with is how to most appropriately process and represent events in real-time. To address these issues implies algorithms that can process information very quickly and a user interface that provides the right compromise between dynamics and interactivity. We feel that subsequent research and experience in the area of real-time information visualization is strongly needed.

At the beginning of our research, we decided to experiment with firewall logs since we were suffering from the information overload syndrome. We learned a significant amount while analyzing these logs with our tools, but we now feel that the content of our firewall logs is poor. We would like systems to provide much more detailed descriptions of each event. To cope with this problem, we are now considering the possibility of merging multiple sources of logs to gain more contextual information. At this stage, our approach does not properly make use of the context in which an event takes place. In fact, it considers each event as a separate entity, while taking into account the activity that has taken place before, or eventually after, is certainly of tremendous importance.

In our firewall example, we mainly relied on information containing categorical attributes. There is still too little information design experience to cope with the representation of data that do not fit on a continuous scale or where no particular order makes sense. Our research would greatly benefit from new visualization techniques, especially if they handle sparse categorical datasets.

Currently, help is provided for actively exploring and passively monitoring event logs. However, one would certainly like to automatically carry out actions to affect the activity, such as redirecting overloaded or faulty services to other hosts, disconnecting intruders, or triggering actions by mimicking previous user behaviors. Even if such automatic responses may not have an elegant theoretical foundation, they may nonetheless be of practical interest. In the same spirit, we would like monitoring tasks to be performed collaboratively, but there is currently no integrated tool to support that.

Conclusion

During this work, we have built tools to monitor, explore, and analyse sources of real-time event logs.

Through the use of self-organizing algorithms, the classification of the event logs is performed automatically and does not rely on any apriori knowledge about the content of the logs. We use a map metaphor which provides the user with a frame of reference and the visual tools which enable an intuitive and effective interpretation of the information contained in the map.

Our tool relieves system and network administrators from the laborious task of scanning long log files and building sets of rules for automatic classification. We also expect the users of our tools to discover hidden patterns of activity in their systems.

While the set of components we developed is still in its infancy, we have nonetheless proven that the approach of interweaving machine learning algorithms and information visualization techniques is potentially a powerful approach for anybody confronted with the analysis of activity-based information.

Acknowledgments

We are indebted to a number of people who have contributed in this research. We are especially grateful to Timothy Jones, Vuk Ercegovic, Florian Raemy, Hans-Peter Frei, Kan Zhang, Jan Schultheiss, Matthew Chalmers, Pamela Cotture, Melissa Binde, and Aline Chabloz, for their interesting discussions, corrections, suggestions, developments, and support. Thanks also to our LISA '98 reviewers for their insightful comments and suggestions.

Author Information

Luc Girardin is currently research staff member and system administrator at the UBS IT research department, Ubilab. His research focus on information visualization, collaborative virtual environments, and adaptive and reliable systems. He has worked as researcher, system administrator, or developer for six years, and holds a masters degree in computer science and telecommunications. He can be reached via email at Luc.Girardin@ubs.com.

Dominique Brodbeck received his PhD in Physics in 1992 from the University of Basel, Switzerland. He then joined the IBM Almaden Research Center (San Jose, CA), first as a post-doc and later as a research staff member to work in areas such as scientific visualization and information visualization for data mining. In 1996 he moved back to Switzerland to join Ubilab and now works on visualization of abstract data, information design, information management, and related areas. Dominique can be reached via email at Dominique.Brodbeck@ubs.com.

References

- [Ahlberg, et al., 1995] Ahlberg, C.; Wistrand, E. IVEE: "An Information Visualization & Exploration Environment." *Proc. IEEE Information Visualization '95*, Atlanta, Georgia, USA, October 30-31, 1995, pp. 66-73.
- [Brodbeck, et al., 1997] Brodbeck, Dominique; Chalmers, Matthew; Lunzer, Aran; Cotture, Pamela. "Domesticating Bead: Adapting an Information Visualization System to a Financial Institution." *Proc. IEEE Information Visualization '97*, Phoenix, Arizona, USA, October 20-21, 1997, pp. 73-80.
- [Chalmers, 1996] Chalmers, Matthew. "A Linear Iteration Time Layout Algorithm for Visualising High-Dimensional Data." *Proc. IEEE Visualization '96*, San Francisco, California, USA, October 1996.
- [Chen, et al, 1998] Chen, Hsinchun; Nunamaker, Jay Jr.; Orwig, Richard, and Titkova, Olga. "Information Visualization for Collaborative Computing." *IEEE Computer*, August 1998, pp. 75-82.
- [Couch, et al., 1996] Couch, Alva L. "Visualizing Huge Tracefiles with Xscal." *Proc. 10th Systems Administration Conference (LISA'96)*, Chicago, IL, USA, September 29-October 4, 1996, pp. 51-58.
- [Frank, 1994] Frank, Jeremy. "Artificial Intelligence and Intrusion Detection: Current and Future Directions." *Proc. 17th National Computer Security Conference*, 1994.
- [Geer, et al., 1997] Geer, Dan (editor); Oppenheimer, David L.; Wagner, David A. and Crabb, Michele D. "System Security: A Management Perspective." Berkeley: *The Usenix Association*; 1997. ISBN: 1-880446-85-5
- [Hofmeyr, et al., 1998] Hofmeyr, Steven A.; Forrest, Stephanie, and Somayaji, Anil. "Intrusion detection using sequences of system calls." *Journal of Computer Security*, 1998 (In press).
- [Hughes, 1996] Hughes, Doug. "Using Visualization in System and Network Administration." *Proc. 10th Systems Administration Conference (LISA'96)*, Chicago, IL, USA, September 29-October 4, 1996, pp. 59-66.
- [Inselberg, 1985] Inselberg, Alfred. "The plane with parallel coordinates." *The Visual Computer 1*. Springer, 1985, pp 69-91.
- [Karam, 1994] Karam, Gerald M. "Visualization using Timelines." *Proc. 1994 International Symposium on Software Testing and Analysis*, Seattle, WA, USA, August 17-19, 1994.
- [Kaski, 1997] Kaski, Samuel. "Data Exploration Using Self-Organizing Maps." *Espoo: Helsinki University of Technology*, 1997.
- [Kohonen, 1995] Kohonen, Teuvo. *Self-organizing maps*. Berlin; Heidelberg; New-York: Springer, 1995. ISBN: 3-540-58600-8.
- [Lankewicz, et al., 1997] Lankewicz, Linda B.; Srikanth, Radhakrishnan, and George, Roy. "Anomaly Detection using Signal Processing and Neural Nets." *Proc. ONDCP International Technology Symposium*, Chicago, USA, 1997.

- [Levkowitz, 1997] Levkowitz, Haim. *Color Theory and Modeling for Computer Graphics, Visualization, and Multimedia Applications*. Boston, Dordrecht, London: Kluwer; 1997. ISBN: 0-7923-9928-5.
- [Li, et al., 1995] Li, Sofianto; Vel, Olivier de, and Coomans, Danny. "Comparative analysis of dimensionality reduction methods," *Learning from Data: Artificial Intelligence and Statistics V*, New York: Springer-Verlag, 1995, pp. 323-331.
- [Tan, 1997] Tan, Kymie. *The Application Of Neural Networks to UNIX Computer Security*. 1997.

Mailman: The GNU Mailing List Manager

John Viega – Reliable Software Technologies
Barry Warsaw and Ken Manheimer – Corporation for National Research Initiatives

ABSTRACT

Electronic mailing lists are ubiquitous community-forging tools that serve the important needs of Internet users, both experienced and novice. The most popular mailing list managers generally use textual mail-based interfaces for all list operations, from subscription management to list administration. Unfortunately, anecdotal evidence suggests that most mailing list users, and many list administrators and moderators are novice to intermediate computer users; textual interfaces are often difficult to use effectively.

This paper describes Mailman, the GNU mailing list manager, which offers a dramatic step forward in usability and integration over other mailing list management systems. Mailman brings to list management an integrated Web interface for nearly all aspects of mailing list interaction, including subscription requests and option settings by members, list configuration and Web page editing by list administrators, and post approvals by list moderators. Mailman offers a mix of robustness, functionality and ease of installation and use that is unsurpassed by other freely available mailing list managers. Thus, it offers great benefits to site administrators, list administrators and end users alike. Mailman is primarily implemented in Python, a free, object-oriented scripting language; there are a few C wrapper programs for security.

Mailman's architecture is based on a centralized list-oriented database that contains configuration options for each list. This allows for several unique and flexible administrative mechanisms. In addition to Web access, traditional email-command based control and interactive manipulation via the Python interpreter are supported. Mailman also contains extensive bounce and anti-spam devices.

While many of the features discussed in this paper are generally improvements over other mailing list packages, we will focus our comparisons on Majordomo, which is almost certainly the most widely used freely available mailing list manager at present.

Introduction

Electronic mailing lists often have humble beginnings: someone collects a list of email addresses of like-minded people, and these people begin sending email to each other using an explicit distribution list. This type of simple list is fairly easy for a novice to start, and in fact many end-user mail applications let people easily set up such distribution lists.

Often however, such mailing lists will grow and evolve, gaining and losing members while existing members' addresses change over time. As they do, explicit lists of addresses become extremely unwieldy. List administrators quickly tire of adding and removing subscribers manually, and answering email pertaining to the list. As a result, administrators generally turn to mailing list management software to automate the process.

The first generation of mailing list managers automated tedious administrative functions such as subscribing and unsubscribing from mailing lists, as well as many of the other common requests, such as getting background information on a list, and getting a list of subscribed members. They also allowed for lists

to be administered via an email interface, so that list administrators would not need to have direct access to the machine on which the list software ran.

However, this generation of mailing list management software has traditionally been quite complex; users are often unable to figure out how to get on or off a list, leading to many messages along the lines of "please unsubscribe me." List administrators often find it time consuming and difficult to perform administrative tasks by email, especially when editing special message headers is required, as is the case with approving held messages in Majordomo. In fact, many of the most popular mail user agents (MUAs) of today (including the Netscape mail reader) make it fairly difficult for the user to edit arbitrary headers. System administrators frequently have a difficult time setting up such software, especially when many commonly desired features such as list archiving are only available as third-party add-ons, if at all.

Mailman is helping to pioneer the second generation of free mailing list managers. While even three years ago email messages were the only reasonable user interface that would make mailing lists accessible

to every Internet user, today the World Wide Web is generally considered ubiquitous. In fact, the Web offers a high level of familiarity and usability for mailing list users, who are typically at least as experienced, if not more so, at browsing the Web. Considering the frequency with which most users interact with the administrative interface of a mailing list, using a Web form that presents all the options is much less of a burden than having to learn or relearn an arcane syntax for mail commands. Ironically enough, instructions for interacting with mailing lists are commonly found on Web pages.

Functionality Overview

Mailman's primary distinction from other mailing list managers is its Web interface, which is discussed in the following section. However, in addition to having all of the features people expect from a list management system, such as digests and moderators, Mailman integrates a rich set of general-purpose features.

One such feature is automatic bounce handling. Much like the SmartList mailing list manager [Sma98], Mailman looks at all delivery errors, and uses pattern matching to figure out which email addresses are bouncing. By default, once the number of bounces from an address reaches a configurable threshold, the address becomes disabled, but not removed. The administrator is then sent a message and can decide whether the address should be re-enabled or removed. However the administrator could set the list to be more aggressive, automatically removing addresses after a certain number of bounces.

We have examined several thousands of bounce messages received while administering Majordomo-based lists, from which we determined the current set of patterns.¹ Applying these patterns to bounces has a two-fold benefit: we do not need to answer "-request" mail, and we rarely need to handle bounce disposition manually. On large lists, this automation can be important, as bounced email can easily produce 10 to 100 times as much email as actual list submissions [Lev97].

Mailman also contains several anti-spam devices that significantly reduce the amount of spam that reaches end users. First, member addresses are not presented in a form that traditional spammer-launched webcrawlers will recognize. For example `mailman_at_list.org` would be used in href links, while in displayed text, spaces would replace the `_at_`.

Second, Mailman's delivery scripts apply a number of configurable and extensible filters to the incoming message, such as requiring the list address to be

named in the To: or Cc: fields, or rejecting messages from known spam sites. These, as well as other measures, have proven to be very effective in preventing most spam from reaching the list, while still allowing valid messages to propagate.

Mailman also offers integrated support for many things that have traditionally been provided in add-on packages, or have required hacking with other list management software. Mailman is distributed with such features as archiving of messages sent to a list, fast bulk mailing by multiplexing SMTP connections, multi-homing for virtual domains and gating mail to and from NNTP news groups. Mailman also uses the GNU autoconf tool to make the setup process easy; in contrast, the Majordomo maintainers admit that Majordomo is difficult to install [Bar98].

Thus, Mailman is able to provide a system administrator with a mailing list manager that is not only easy to install, but also is easy to use at every level, and includes the major pieces of functionality a list administrator might want without requiring additional searches and downloads.

Web Interfaces to Mailing Lists

While Mailman does provide Majordomo-like mail-based commands for compatibility, we downplay this, as we feel that a good Web-based user interface is much more desirable to the majority of users. Our Web-based interface allows for full access to all of Mailman's features, including subscription and option requests, browsing lists on the same (potentially virtual) host, viewing Web-based Hypermail-like archives, etc.

There are many third-party Web front-ends to Majordomo [Bar98]. However, most of them are little more than simplistic interfaces to subscribing and unsubscribing. The most notable exception is MajorCool [Hou96], which additionally provides end users with a way of browsing all mailing lists on a machine, as well as a full-featured interface to the list configuration. However, MajorCool suffers from several usability problems, all of which are addressed by Mailman.

First, MajorCool has the problem that malicious users can subscribe and unsubscribe other people from mailing lists over the Web. Mailman, on the other hand, requires confirmation emails for subscriptions. For unsubscribing, users must enter a password into a CGI field, which can be generated by Mailman, and delivered to the subscribed email address on request.

Second, MajorCool requires that it and an HTTP server must be co-located on the machine running Majordomo and Sendmail [Hou98]. In contrast, Mailman has been tested with a mail transport and Web server running on separate machines in an NFS environment, and has been tested with the transport, Web server, and Mailman all running on separate machines, where Mailman scripts are run via rsh or ssh.

¹Bounce patterns are based on regular expressions, and are not currently extensible without editing the Mailman source code.

Third, MajorCool's interaction with end users is limited. Its goal with respect to end users is to give them a way to browse all the lists on a machine, not to provide a nice Web-based mechanism for interacting with the mailing list. Mailman provides full support for editing options such as the digest mode on both a per-list and per-user basis and whether posts to a list should be sent back to the user. List member email addresses can also be kept completely private by suppressing their visibility on the subscriber list Web page.

Finally, MajorCool's administrative interface is mainly geared towards interfacing to the traditional Majordomo configuration. In contrast, many of Mailman's administrative options allow for customization

of the list's Web interface. In fact, Mailman also allows the list administrator to provide a "real" Web page for his mailing list, and he can edit HTML templates for this page via a password protected Web-based interface. MajorCool essentially lacks the notion of each list having its own home page.

Example

Figure 1 shows a screen shot of part of the Web subscription and general list information page for a Mailman mailing list.² All of the presented

²This and other screenshots in this paper were generated by Mailman 1.0b4. Some of the details may have changed since the time of writing.

Mailman-Users -- Mailman maillist management system forum

About Mailman-Users

This maillist is for users and other parties interested in the *mailman* maillist management system. There is a separate maillist for people interested in discussion about development of the system - [mailman-developers](#).

To see the collection of prior postings to the list, visit the [Mailman-Users Archives](#).

Using Mailman-Users

To post a message to the all the list members, send email to mailman-users@python.org.

You can subscribe to the list, or change your existing subscription, in the sections below.

Subscribing to Mailman-Users

Subscribe to Mailman-Users by filling out the following form. You will be sent email requesting confirmation, to prevent others from gratuitously subscribing you. This is a public list, which means that the members list is openly available (but we obscure the addresses so they are not easily recognizable by spammers).

Your email address:

You must enter a privacy password. This provides only mild security, but should prevent others from messing with your subscription. Do not use valuable passwords! Once a month, your passwords will be emailed to you as a reminder.

Pick a password:

Reenter password to confirm:

Would you like to receive list mail batched in a daily digest? ☒ No ☐ Yes

Mailman-Users Subscribers

Click here for the list of Mailman-Users subscribers:

To change your subscription (set options like digest and delivery modes, get a reminder of your password, or unsubscribe from Mailman-Users), *either* enter your subscription email address:

... *or* select your entry from the subscribers list (see above).

Figure 1: Web subscription and general list information page.

components are configurable by the list owner, including the list description shown in the title banner, as well as the HTML displayed in the "About" section. While these are easily changed by setting options on the list administration page, in fact the list owner can actually edit the full HTML template from which this page is generated. Thus the list owner can rearrange sections, and even omit standard boilerplate text, such as might be necessary if a list was configured not to provide archives, or if postings were completely disabled.

Note that when subscribing, a user must pick a password. This password is used by members when they change their subscription options. Password reminders are periodically mailed to members.

Subscribing users also have the option of receiving messages as they are delivered to the list, or batched in digest form. The list owner can enable or disable digests on a per-list basis, and set other digest parameters. Of course, users can easily switch from receiving individual postings to receiving digests via their personal options Web page. This is useful for when a user goes on vacation and wants to continue to receive mailing list traffic, but wants the impact on their mailbox to be minimized.

The general information page also contains buttons to view the list of subscribers (for public lists; individual members can still opt to remain unpublicized), and to edit an existing member's list options.

Architecture

Mailman is written almost completely in Python [Pyt98], a freely available, object-oriented scripting language. There are a few C wrapper programs for security purposes. Mailman currently requires at least Python 1.5, which is freely available in both source and (for many platforms) binary form at <http://www.python.org/>.

System Architecture

The Mailman system architecture is illustrated in Figure 2. In the center of the system are the core Mailman classes and modules, organized as a Python package [Ros97]. The architecture of these classes is described in the next section. There are two sub-packages in the core package, one that contains classes for logging, and another that contains modules that support the CGI interface.

The Mailman package mediates access to various disk files used during its operation. For example, the logging classes write update messages to file when subscriptions or unsubscriptions are requested or fulfilled, or when various types of error conditions occur. Lock files are created and consulted by package modules for synchronization between processes. Also, as described in more detail below, every active list is associated with some persistent state, contained in list database files.

For increased security, subscription requests that originate via the Web interface are held for confirmation by the subscribing email address. These pending confirmations are also contained in files on disk, as are other pending actions, such as postings that are being held for approval. When a user subscribes via the Web, he is emailed a confirmation message containing a random number. The user need only reply to the original message in order to be subscribed. This feature eliminates the possibility that users will be subscribed to mailing lists against their will, while imposing minimal burden on the user. The list owner has control over the confirmation mechanism used as well.

Templates are used for most of the textual messages that are generated by Mailman and sent to list members via email. This has one immediate and one future benefit. First, by removing most of the textual messages from the source code, it is easier to maintain and modify the messages, with systematic approaches for including placeholders in the template. Second, this arrangement provides the framework for future localization efforts. Although not currently implemented, this framework would allow us to arrange the templates in language specific subdirectories, for access on a per-list or possibly per-user basis.

The various front-end mechanisms used to access Mailman functionality are shown at the top of the figure. On the left is shown access through the incoming mail system; Mailman supports several mail transport agents (MTAs), including sendmail and qmail. In a sendmail installation for example, aliases are installed in the system's `/etc/aliases3` file. Typically, five aliases are installed for each active mailing list. Three of these point to a C wrapper program, which in turn executes Python code to perform various email-based commands such as posting a message to the list, evaluating Majordomo-style list commands sent to the "-request" address, or forwarding a message to the list owner.

The most common access method is through the Web interface, as shown on the top right of the figure. Here, the user or list administrator views one of the various Mailman Web forms in their browser, entering information in the text entry fields and/or clicking buttons presented on the form. When the form is submitted, the browser posts it to the Web server, which can be any standard Web server configured to run CGI scripts. The CGI script is another C wrapper program that in turn calls a central Python "driver" script. The driver then imports the appropriate module from the CGI support package and executes it for the selected functionality.

³This file may in fact reside in other locations, depending on the system. For example, on many Solaris machines this file is located in `/etc/mail/aliases`.

The driver script's primary function is to catch and usefully report any error in the Mailman system. Normally such errors would generate Python exceptions, which if left uncaught, would percolate up to the top of the script's execution stack, and cause the CGI script to exit with an error code. This in turn would force the HTTP server to display a less than useful error message to the end user. The driver script is designed to catch all errors and to report the most useful error message possible. When such an error occurs, the end user is presented with a Web page informing them of the error, including a Python traceback and a dump of the CGI environment variables. This information is also written to a Mailman log file on the list management site. In this way, such errors can be quickly identified, and end users are given more information than just a generic Web server failure message.

Another mechanism shown in Figure 2 is access via cron jobs. Mailman contains a number of cron scripts which are used, among other things, to mail the periodic password reminders. These cron scripts use the same core Mailman classes as other subsystems previously described.

Mailman also contains a number of scripts intended to be run by the system administrator via a shell command line. These scripts use the core package to provide higher level functionality. For example, to create a new mailing list, the system administrator would execute the `newlist` command, providing the name of the new mailing list, the list administrative

password, and the email address of the list owner. This is all that is necessary to create the list; all other list configurations are performed through the Web administrative interface. Other command line scripts are provided to set the site password, remove lists, subscribe members en masse, etc.

One of the more unique features of Mailman is that the core classes can be accessed interactively via the Python interpreter. This allows the system administrator to simply fire up an interactive Python session, import the appropriate Mailman module from the package, instantiate instances of various classes, call methods on those instances, and even inspect the various objects involved.

This is an extremely powerful ability, because it means that the system administrator is not limited to those functions which are provided by the various Mailman scripts. In fact, the administrator proficient in Python can easily code their own routines using the core classes, prototyping and developing them by using an interactive Python interpreter session. The administrator is even able to perform one time procedures directly inside the interpreter.

It is even conceivable that other access mechanisms and front-ends could be created. For example, more specialized non-Web based GUIs could be developed, or perhaps a set of CORBA interfaces to the Mailman system could be specified. This might be useful, for example, to a user that is a member of a dozen or so mailing lists running on many systems

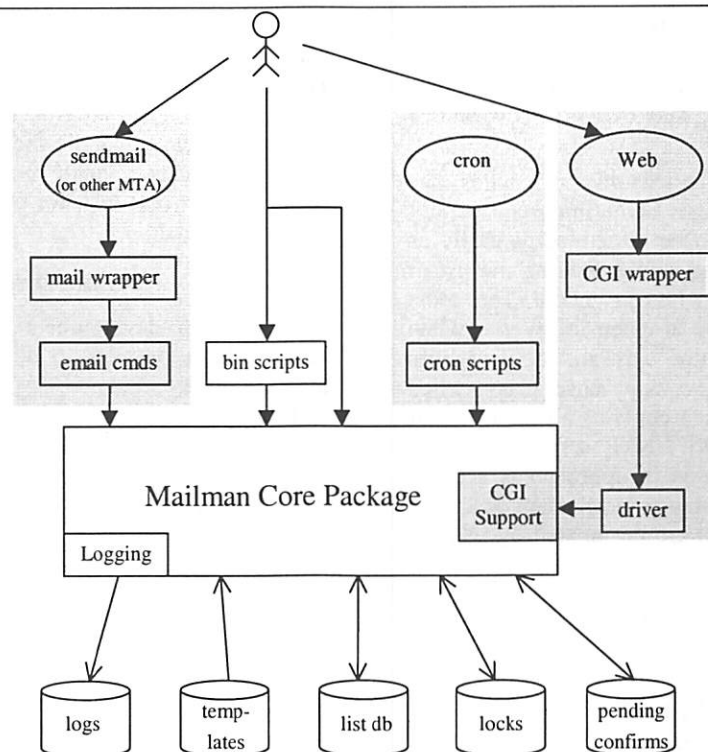


Figure 2: System Architecture.

throughout the Internet. Having a CORBA interface to Mailman would allow such a user to write a single script (in his language of choice) which could switch his subscription to digest mode when he goes on vacation, and then back to his preferred distribution mode upon his return.

Software Architecture

The central component of the Mailman core package is the `MailList` class, instances of which are used to represent every active mailing list. Instance variables ("attributes" in Python parlance) contain all the information pertinent to the mailing list, including member addresses and option settings. This information is stored in a persistent database via Python's built-in object serialization mechanism.

Thus, for example, when a user accesses a particular mailing list via the Web, the invoked CGI script instantiates the `MailList` class, passing to the constructor the name of the mailing list. When created, the instance variables for this object are restored from the persistent database. Mailman uses Python's `marshal` module [Py98A] to save and restore persistent attributes. `marshal` is a low-level built-in module providing object serialization. The higher level `pickle` module is not used since the data structures involved are relatively simple, and `marshal` thus provides better performance.

The `MailList` class is multiply derived from several task-oriented mix-in classes. These mix-in classes provide the basic mailing list-centric functionality described in the previous sections, such as the ability to handle Majordomo-style email commands, generate HTML content for Web presentation, perform digesting, archiving, and delivery, and handle bounce disposition, etc.

The use of task-oriented mix-in classes has advantages and disadvantages. One important benefit is that new tasks can often be integrated as easily as creating a new mix-in class, and extending the list of base classes for the `MailList` class. The most recent example of this ease of extensibility was when the Usenet gateway feature was added. This was implemented by creating a new base class called `GatewayManager`, which contains all the code for posting email messages to NNTP servers. Another important benefit of the mix-in approach is seen in conjunction with the persistency mechanism described above. Persistent attributes are designated by using a naming convention; specifically, if the attribute name starts with an underscore it is not persistent. Python's introspection capabilities allow Mailman to inspect all the attributes of an instance, ignoring those with names beginning with an underscore. The remaining attributes are stored in a Python dictionary, and that dictionary is then saved to disk with `marshal`.

When a new mix-in base class is added, and that class adds new attributes to the state of the list

instance, those attributes are automatically made persistent due to this introspection property. Of course, there are versioning issues to deal with, but simply by adhering to the naming convention described above, new state supporting new functionality can easily be added.

One disadvantage of the mix-in architecture is that it can complicate the interactions between the tasks. Primarily, experience has shown that simply initializing each base class's attributes can be tricky.

Many persistent attributes are tied to options presented on a Web page. Figure 3 shows one of the list administration pages for a Mailman list. Shown here are some of the list specific privacy options available, including whether the list is advertised and what style of subscription confirmation is to be used. Each of these options is coupled to an attribute on the `MailList` instance for the specified list. When the option is changed on the posted Web form, the instance attribute is modified, and the state is saved on disk.

Performance

While Mailman is too new to have much hard data in the way of performance metrics, we do know that, given a well designed mailing list management system, the performance of the mail transport agent (MTA) will have a much more significant impact. We have found that even a low-end configuration can handle large amounts of traffic. For example, one mailing list managed by Mailman has had up to 3000 subscribers, and often receives 100 messages in a day (i.e., hundreds of thousands of daily deliveries). The list runs on a low-end Pentium with 48MB of RAM. The machine runs sendmail on GNU/Linux. The machine also hosts an NNTP news feed for a small ISP, and is able to handle the load, although sendmail sometimes needs to queue messages. As Mailman proceeds through beta test, we plan to gather more detail performance data.

Future Directions

Mailman development is ongoing and highly active. Major projects to be undertaken in the near future include:

- Integrating searching with list archives.
- Manually configurable and automatically used relays for distributing server and network load (along the lines of RFC 1429 [Tho93]).
- An optional threaded persistent server, as opposed to the current "start-by-request" model shared with Majordomo.
- A separation of the roles of list administrator and list moderator.
- PGP integration.

Availability and Compatibility

Mailman 1.0 is currently in beta release, but is already being used at a number of sites. More

information on Mailman can be had at <http://www.list.org>. Various mailing lists are currently being run for Mailman discussions (managed by Mailman of course!):

- URL <http://www.python.org/mailman/listinfo/mailman-users> is for system administrators who are using Mailman to manage their mailing lists.
- URL <http://www.python.org/mailman/listinfo/mailman-developers> is for those who would like to help future development of Mailman.

Bleeding edge snapshots of the Mailman development code is also available via anonymous CVS. See the developers URL above for details.

Mailman should work out of the box on any Unix-based platform on which Python runs. It is known to work on SunOS, Solaris, all major distributions of Linux, FreeBSD, Irix and NextStep. Mailman will work with any MTA, since it communicates via the SMTP port instead of through a command. However, Mailman currently generates sendmail-style aliases only. Therefore, aliases for MTAs such as qmail must be modified and installed by hand. Python itself has been ported to a large number of systems, including most known Unix-like systems, various Windows platforms (NT and Windows 95), and MacOS. Python source code is freely available, as are pre-built binaries for many platforms.

The screenshot shows a Netscape browser window titled "Mailman-Users Administration - Netscape". The address bar shows the URL <http://www.python.org/mailman/admin/mailman-users/privacy>. The page content is titled "Mailman-Users Maillist Configuration - Privacy Options Section".

Under "Configuration Categories", there is a list of links: [General Options](#), [Membership Management](#), **=> Privacy Options <=**, [Regular-member \(non-digest\) Options](#), [Digest-member Options](#), [Bounce Options](#), [Archival Options](#), and [Mail-News and News-Mail gateways](#).

Under "Other Administrative Activities", there is a list of links: [Tend to pending administrative requests](#), [Go to the general list information page](#), and [Edit the HTML for the public list pages](#).

A note states: "Make your changes, below, and then submit it all at the bottom. (You can also change your password there, as well.)"

The main section is titled "Privacy Options". A sub-note says: "List access policies, including anti-spam measures, covering members and outsiders. (See also the [Archival Options](#) section for separate archive-privacy settings.)"

The form is organized into sections with headers:

- Subscribing**
 - Advertise this list when people ask what lists are on this machine? ☐ No ☒ Yes
 - Are subscribes done without admins approval (ie, is this an open list)? [\(Details\)](#) ☐ No ☒ Yes
 - What confirmation is required for on-the-web subscribes? [\(Details\)](#) ☐ None ☒ Requestor confirms via email ☐ Admin approves
- Membership exposure**
 - Who can view subscription list? [\(Details\)](#) ☒ Anyone ☐ List members ☐ List admin only
 - Show member addrs so they're not directly recognizable as email addrs? [\(Details\)](#) ☐ No ☒ Yes
- General posting filters**
 - Must posts be approved by a moderator? [\(Details\)](#) ☒ No ☐ Yes

At the bottom of the form area, it says "Document: Done".

Figure 3: List administration page.

Mailman should work with any HTTP daemon that allows for CGI directories. It is known to work with Apache, NCSA, and Java Web Server.

For current Majordomo users, the transition to Mailman is straightforward; there is a command-line script in the distribution that imports a Majordomo distribution list into Mailman.

Acknowledgements

Mailman was originally written by John Viega. It has since been extended, and is currently being developed and maintained by John Viega, Ken Manheimer, and Barry Warsaw. The mailman-developers mailing list and the Python community have provided invaluable feedback on this software, including Guido van Rossum, Scott Cotton, Janne Sinkkonen, Michael McLay and Hal Schechner. We would like to thank these people and all others on the Mailman users and developers lists.

Mailman uses free software by Timothy O'Malley for dealing with HTTP cookies. It also integrates Pipemail, free software by Andrew Kuchling that handles message archiving. The archiving code also uses free software by Aaron Watters.

We would like to give special thanks to the Python Software Activity (PSA), and the Corporation for National Research Initiatives (CNRI) for hosting the PSA. We would also like to thank Guido van Rossum for inventing Python.

We would also like to give special thanks to Richard Stallman and the Free Software Foundation for their support and guidance.

Author Information

John Viega is a Research Associate at Reliable Software Technologies. He holds an M.S. in Computer Science from the University of Virginia. His research interests include software assurance, programming languages, and object-oriented systems. Contact him at viega@rstcorp.com.

Barry Warsaw is a systems engineer at CNRI. He is member of the team developing advanced Internet technologies such as the Knowbot Operating Environment mobile code system, the Application Gateway System high-availability server farm, and the Grail Internet Browser. He is a member of the Python Software Activity and contributes to the development of Python. He has been involved with various open source projects for many years. Contact him at bwarsaw@cnri.reston.va.us.

Ken Manheimer is a member of the technical staff at CNRI, developing and researching application of mobile agent systems, server farms, and other advanced network technologies. His former life involved managing a large installation of Unix systems at NIST, where he devised, with Barry Warsaw, the Depot for sharing installed software across sites –

which he presented many LISA's ago (LISA IV, 1990). Currently Ken manages only a few systems, including the python.org server system, on which he manages the Python Software Activity. Contact Ken Manheimer at klm@cnri.reston.va.us.

References

- [Bar98] D. Barr. *The Majordomo FAQ*. <http://www.greatcircle.com/majordomo/majordomo-faq.html>.
- [Cha92] D. Chapman. "Majordomo: How I Manage 17 Mailing Lists Without Answering '-request' Mail." *Proc. Usenix LISA VI*, Oct. 1992.
- [Hou96] B. Houle. "MajorCool: A Web Interface To Majordomo." *Proc. Usenix LISA X*, Oct. 1996.
- [Hou98] B. Houle. *MajorCool Introduction*. <http://ncrinfo.ncr.com/pub/contrib/unix/MajorCool/Docs>.
- [Lev97] J. Levitt. *Ten Questions For Majordomo (An Interview With D. Brent Chapman)*. <http://techweb.cmp.com/iw/author/internet8.htm>.
- [Ros97] G. van Rossum. *Built-in Package Support in Python 1.5*. <http://www.python.org/doc/essays/packages.html>.
- [Pyt98] *The Python Language Website*. <http://www.python.org/>.
- [Pyt98A] *Built-in Module marshal*. <http://www.python.org/doc/lib/module-marshal.html>.
- [Sma98] *The SmartList FAQ*. April 1998 revision. <http://www.mindwell.com/smartlist/>.
- [Tho93] E. Thomas. *RFC1429: Listserv Distribute Protocol*. Feb. 1993. <http://www.faqs.org/rfcs/rfc1429.html>.

Drinking from the Fire(walls) Hose: Another Approach to Very Large Mailing Lists

Strata Rose Chalup, Christine Hogan, Greg Kulosa, Bryan McDonald, and Bryan Stansell – Global Networking and Computing, Inc.

ABSTRACT

This paper describes a set of tools and procedures which allow very large mailing lists to be managed with the freeware tool of the administrator's choice. With the right approach scaling technology can be applied to a list management tool transparently.

In recent years, many ingenious methods have been proposed for handling email deliveries to mailing lists of several thousand subscribers. Administration of a mailing list is not limited to message delivery, however. Tasks such as managing subscribers, dealing with mail bounces, and preventing list spamming also become more difficult when applied to very large lists.

As a case study, this paper describes the process of moving the well-known "Firewalls" mailing list from its original home at GreatCircle Associates to a new infrastructure at GNAC. The process was thought to be straightforward and obvious, and it soon became apparent that it was neither. We trust that our discoveries will benefit other systems administrators undertaking similar projects, either concerning large mailing lists or moving complex "legacy systems."

Introduction

"And you may ask yourself,
Well ... How did I get here?"
– Talking Heads

The Firewalls list began in 1992, at GreatCircle Associates. It quickly evolved into an important forum for new ideas, in-depth technical discussions, and impassioned flame wars. Eventually it would encompass roughly 4500 real-time subscribers and 4900 digest subscribers. A large number of total subscribers were "exploder" or reflector lists passing Firewalls list traffic to unknown third parties at companies and universities around the world.

Daily message counts ranging from a norm of 20 to peaks of 75 or more yield message deliveries of 95,000 to 342,400. In addition, a growing problem with spammers began raising both list traffic and the collective blood pressure of subscribers and list administrators.

In the fall of 1997, list founder Brent Chapman joined a startup company as a key player and realized that he would have little attention left over for anything else. In his own words:

"Firewalls and Firewalls-Digest are very popular, high-volume mailing lists, and they take a fair amount of time and effort to maintain. Life in a high-profile Silicon Valley startup doesn't leave much time for anything else, though, so Great Circle Associates is going into hibernation. Therefore, after five and a half years and 111+ Mbytes of discussions spanning 45,517 messages and 3018 digests, the Firewalls and Firewalls-Digest mailing lists are

moving to a new home at GNAC, which is a consulting and managed services firm based here in the Silicon Valley that I think highly of." [0]

Given the explosive growth of the list over the years, and the demands on Brent's time, it is very much to his credit that the list was still functional up to that point. Now it was GNAC's turn to re-examine the list and find out how to bring it back up to speed.

The Existing System

"Like a crystal cathedral afloat on the tide
comes a mountain of ice
on the course to collide,
while passengers sleep thinking
God's on their side..."
– Peter Schilling,
"Terra Titanic"

The Firewalls list environment that GNAC inherited turned out to be, as we expected, a complex system of many moving interdependent parts.

We quickly discovered that the core of the Firewalls list was the expected Majordomo list manager [1] wrapped around a dual-sendmail queueing structure.

To optimize the handoff between majordomo and sendmail, Brent had set up a special outbound queue area for list traffic. The `sendmail_command` and `sendmail_command_flags` in `majordomo.config` were modified to implement a queue-only sendmail [2] in a custom queue area.

In addition to the basic Majordomo processing of the lists, part of the "Firewalls list" functionality was

providing archives via FTP and HTTP. The Mhonarc [3] text to HTML converter and some scripting glue took care of the web-accessible archives, and scripts regularly copied standard Majordomo archives into an FTP hierarchy.

There were a number of watchdog scripts to warn about majordomo list processing malfunctions (e.g., list truncation), as well as some behind the scenes scripting that created a meticulous "clean archive" of the list postings for paranoia's sake.

While there were certainly intricacies, we could see that the basic structures were sound, and working well enough on the Great Circle server.

Where Angels Fear to Tread

We had made some detailed queries about the Great Circle server, looking for load patterns and other duties performed by the machine. We'd chosen a similar (in fact, more heavy-duty) server for our purpose and felt confident that it could handle anything that the older Great Circle server had been handling.

For the configuration of the server, we decided to make minimal changes to the operating systems and messaging configurations while the list moved. We carefully prepared a tarball from Brent's server, installed it on our host, and did the basic hostname customizations required to make it run. The preliminary tests looked good. Mail queued into the right places, appeared in archive directories, FTP storage, web pages. Digest files grew. We were ready for prime-time.

We arranged a special "test mode" that would simulate list traffic [4] and, with great anticipation, we turned the key. We knew that there was a heavy spam load, and a lot of traffic, but we were on a faster server with more disk spindles, greater memory, and a wider network pipe. Nothing could go wrong. Go wrong. Go wrong. [5]

In the Cold Light of Day

Test messages would never make it out of the queue. The server would chronically lock due to forking problems. There were crashes and lockouts due to memory problems. Well, this is why we test things in the first place.

The dismal failure of the first "production test" caused us to re-consider our stand on keeping the list management machine and software as close to the original as possible. We realized that we needed to go back to "square zero" and examine the fundamental structure of how the message flow worked. After a couple of false starts, detailed in subsequent sections, we soon had messages turning around in record time. At that point we turned our attention to the secondary challenges which we we had inherited with the management of the list, bounce mail and spam. These at least had been the focus of directed planning for future enhancements.

The amount of bounce mail that a list of the size and volume of Firewalls can produce is phenomenal. Interspersed with the bounce mail would be real requests from real people who needed something from the list managers. We had to find some way to ensure that we were responding in a timely manner to these requests without getting overwhelmed by the bounce traffic.

We also knew that the list had become a favorite venue for spammers sending their useless and annoying missives. Stopping the spam while still allowing legitimate posting would pose its own challenges, due not only to the sheer number of subscribers but to the percentage of "subscribers" which were actually mailing lists rather than individuals.

The volume of messages to the list over time is shown in the graph below. Shortly after GNAC took over the list the spam problem reached a critical point,

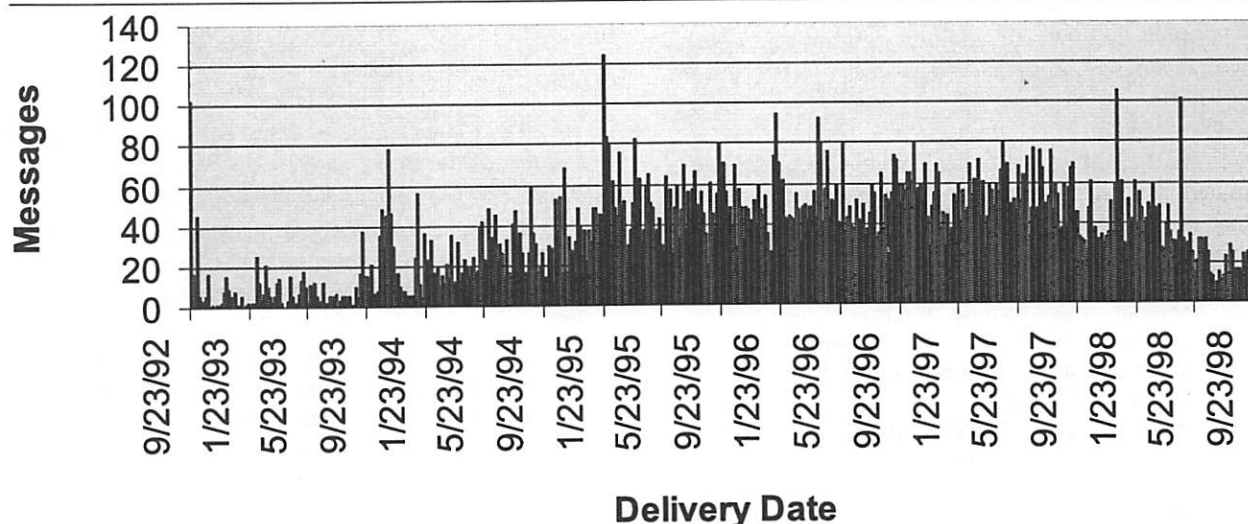


Figure 1: Daily message deliveries.

traffic to the list was high and consisted mostly of spam and complaints about the spam. Immediately after we implemented spam-control measures, list traffic dropped to an uncharacteristic low for a while before resuming more normal levels, but without the spam.

Host Tuning

"Close to the middle of the network,
It seems we're looking for a center.
What if it turns out to be hollow?
We could be fixing what is broken."

— S. Vega,
"The Big Space"

The Great Circle server had been largely untuned, which surprised us in light of the implied message delivery load of the Firewalls and Firewalls-Digest lists. However, in the interest of minimizing changes, we had gone with an "out of the box" BSD/OS config. We had also slavishly copied the configurations of the Firewalls-specific application services in our zeal for compatibility.

As we discovered, our server was doing much more than the original server. As you will see below, the Great Circle server was not actually delivering all the messages to the subscribers, but instead was using relaying services of a large ISP for the actual delivery. Our machine was consequently not tuned correctly to deal with the memory and process usage profile that this task required.

We first became aware that the original machine was relatively un-tuned when we discovered that the sticky bit was not set on the system copy of Perl. From there we did further digging, and decided we needed to go over the entire system and analyze it from scratch. [6,7]:

Here are the major changes we introduced. None of them is necessarily dramatic in impact, but together they represent considerable improvement.

- put operating system and application binaries on different disks
- doubled our swap space
- balanced swap between disks
- set sticky bits on Perl, sendmail, other key system apps
- set sticky bits on all Majordomo binaries & scripts
- rebuilt kernel and upped syslimits.h variables (MAX_CHILD, NPROCS)
- installed a cacheing named [8]

Later we would come to double the physical memory and increase KMEMSIZE to handle some unusual custom processes that we will be describing below.

Message Delivery

"I've been standing here waiting,
Mr. Postman,
so-o-o patiently –
for just a card or just a letter ..."
– The Marvelettes,
"Please, Mr. Postman"

Once we had the machine tuned, we turned to the list processing itself. Message turnaround time had been in the order of days before we took over the list. It was still at that order of magnitude, and when the message volume was high, our outbound queue was growing faster than mail was getting delivered. There was potential for serious backlogs that would cripple the list.

Mail basically wasn't moving. We knew that historically the list was plagued by slow mail, but that the queues didn't back up too badly. Why was our mail backing up? We went back to the Great Circle server to find out. The answer turned out to be our choice of "smart host" in the sendmail-lists.cf file, which we had blithely customized to work with the usual GNAC environment.

As we mentioned earlier, Majordomo was configured to use a queue-only sendmail (designated "sendmail-lists") for message generation. A separate sendmail daemon would process that queue and keep it moving. We discovered that for the sake of expedient message processing by Majordomo, all recipients of the message were packed into a single RCPT line. Those of you who have dealt with sendmail extensively are winning right now, aren't you?

We also discovered that, in a neat private arrangement dating back several years, Brent had arranged for his servers at greatcircle.com to have relaying capabilities through the UUNet mail servers. Thus the Great Circle sendmail-lists.cf file merely specified "mail.uu.net" as the smart host, causing everything to be forwarded to it for processing. Due to the way UUNet round-robins its mail services, this would effectively spread out the processing of these troublesome messages with monstrous RCPT headers.

Of course, GNAC did not have the option of passing those messages to UUNET. We would have to deliver these messages directly, 4K of RCPT addresses or not. While GNAC has an excellent mail infrastructure, their core business does not involve ISP-style mail for thousands of individual subscribers. Thus GNAC did not have the quantity of dedicated mail-delivery resources to simply toss the messages into the network and let them go.

Chunk-Style, Just Like Home-Made

A message with over 4000 recipients can take literally days to deliver. A sendmail instance processing the message will work its way laboriously down the recipient list, pausing at every time-out. It may run out of resources and die, causing a new sendmail to take

up the torch. No problem, you say, since the new sendmail instance can use the xNNNNN queue file to pick up where the old one left off. Yes, but first it has to retry all the "deferred" hosts that timed out. Even if the messages are being farmed out to multiple servers, each individual message is going to reach individual subscribers in a highly non-deterministic fashion.

In analyzing our logs and transfer status files, we found that message deferrals would typically be due to remote name servers or mail servers failing to respond before timeout. Due to the large and diverse population of the list, we would see rather shocking ratios of failures to successful deliveries. Many of those failures were multiple failures trying repeatedly to deliver the same message to the same site.

At an architectural level, we knew that we had to get away from the multi-thousand RCPT lines business. We also knew that we couldn't simply force one recipient per message without making the sendmail queue directory so large that directory search time would become a significant factor. Directories with over 10K nodes are generally undesirable [6] and at over 4K subscribers we would quickly flood the queue directory.

Initially we assumed that we would get our biggest "win" by employing a program such as mailcast [9] to batch and sort the recipients by domain and MX record. Mailcast would simply queue them up and send one nice copy off to the right host and we'd shake each others' hands and go off to hoist a cold one or two. Imagine our consternation when we discovered that in fact out of approximately 5,000 individual subscription addresses, some 4,000 were in fact unrelated by host, domain, or MX record. For the roughly 4,000 digest recipients, we found about 3,500 uniquely unrelated addresses. Ouch. For us, this approach was largely indistinguishable from "one recipient per message."

Since there was not a strong natural grouping between addresses, it seemed that arbitrary recipient chunking would be the way to go. We immediately thought of bulkmail [10], a mail-sending utility that can perform chunking on huge recipient lists. As we looked into the specific configuration of bulkmail, we found that bulkmail and majordomo were not trivially compatible. Majordomo wants to invoke a mail command and send a message to it. Bulkmail wants to read in files with a message and a recipient list. We spent a couple of lunches wrangling over which of them to hack to accept the other's view of the world, and how exactly to structure the changes to minimize future-release porting issues. Any way we looked at it, it looked ugly.

The Portable Queue

At this point, having gone far enough down the rathole to smell cheese, we popped back up into the sunshine to re-examine the original goal, namely chunking the messages into deliverable size. We

realized that we were already producing a clean, queued message with a highly well-defined structure [11] in a place that we could control. There was no reason that we had to perform the chunking at message generation. We could do it right in the queue itself.

Before beginning the move of the Firewalls list, we had done some preliminary mailflow architecture. Our original plan was to move the list without structural changes, then to apply our idealized architecture in careful stages. Based on the production testing, we clearly had to accelerate things quite a bit.

One of the original elements we'd planned to introduce was time-based queuing, where messages are recursively sifted among various queues based on how long they have been pending [12]. Reading up on this approach, we were reminded once again of what every postmaster knows: queue files are portable.

One of the standard postmaster rites of passage is dealing with a major multi-day mail backlog on your bastion host. You eventually realize that the most sane thing to do is to turn off incoming mail, move all the queue files into a holding directory, then turn on mail again. Meanwhile, you do a little quick scripting to sort things based on which internal mailhubs are in the envelope headers, make a few tarballs, and just FTP them down into the right mailhub's queue, unpack, voila! We decided to go one step further and "MIRV" [13] the queue files.

Split Personality

We turned off the "sendmail-lists" invocation of sendmail and replaced it with a cron job called "qsplit." The qsplit Perl script runs every 5 minutes and examines the sendmail-lists queue directory. Each queue file is parsed. The unique portion of the name (e.g., "ABC12345" in "qfABC12345") is stored as Sident and used to generate new qf files. If the number of RCPT lines in the qf file exceeds the qsplit variable \$CHUNKSIZE, the message is processed into multiple messages of \$CHUNKSIZE recipients and zero or one messages of less than \$CHUNKSIZE.

Each new qf file has a sequential number appended to Sident. Thus the first split file from "qfABC12345" would be "qfABC123451," then "qfABC123452" and so on. Since sendmail will generate unique queue file identifiers within a given sendmail queue area, using this method guarantees unique identifiers for split queue files.

Qsplit is also configured to know about an arbitrary number of sendmail queue directories. If the number of recipients in a parsed qf file is less than \$CHUNKSIZE, qsplit will move the message into one of the preconfigured queue directories. A round-robin effect is achieved by keeping track of the last queue directory into which a file has been placed and putting the next one in the subsequent directory, wrapping around as necessary.

For efficiency, each "new" df file is merely an appropriately numbered hard link. This is particularly important for the Firewalls-Digest postings, where the df file can be quite large. Note that since hard links will not work across partitions, the "sendmail-lists" directory and the processing queue directories must be on the same filesystem. Qsplit of course removes the original qf/df files after the splitting is accomplished. This is why we use hard links rather than symbolic links, since hard links have no concept of "the original file."

Splitting the qf files in this way had a dramatic effect on message turnaround time, as the following graph shows. For more than a week before we finally settled upon and implemented our splitting solution, we had been manually splitting qf files and distributing them as described. That period is seen in the graph as a low before a final spike.

Spawn 'til You Die

Each of the queue directories (10, at present) runs separate instances of sendmail. All of the directories are managed by a shared spawning daemon, called simply "spawn.pl". Some experimenting was necessary to find the right timing to use within the configurations, so various copies named things like "slow-spawn" were created for test runs.

The queue management daemon (spawn.pl) spawns as many copies of itself as there are queue areas. It then watches its children and re-starts any of the spawned processes that die. Each of these child spawners is chartered with keeping ten sendmail

daemons running to process its queue area. The child spawners keep track of their sendmail children, restarting a new sendmail whenever one dies.

There is logic built into the spawners to check configurable variables for the load average on the system, and the amount of memory available. If the load is too high, or memory too scarce, the child waits until there are more resources available before starting a new process. The initial spawning of the children processes themselves is also subject to the same limitations. The load average limit in the spawner is set lower than the sendmail threshold, since starting a sendmail will cause a load average spike that might cause sendmail to not do anything once started. In addition, a variable controls the timing between each child spawner or new sendmail, so as to minimize load disruption.

To further speed things up, we also implemented sendmail's host-status feature. This creates a directory structure containing information about when a sendmail last tried to contact a given host, and whether it was up or down. If it is down, sendmail doesn't try that machine again unless the specified re-try timeout has expired. We used the sendmail default of one hour.

The trade-off between the massive amount of disk access that this caused and the saved processing and wait time has proven to be worthwhile. Setting all of the sendmails across all of the spawn-managed queue areas to use the same host-status caching has given us even greater efficiency.

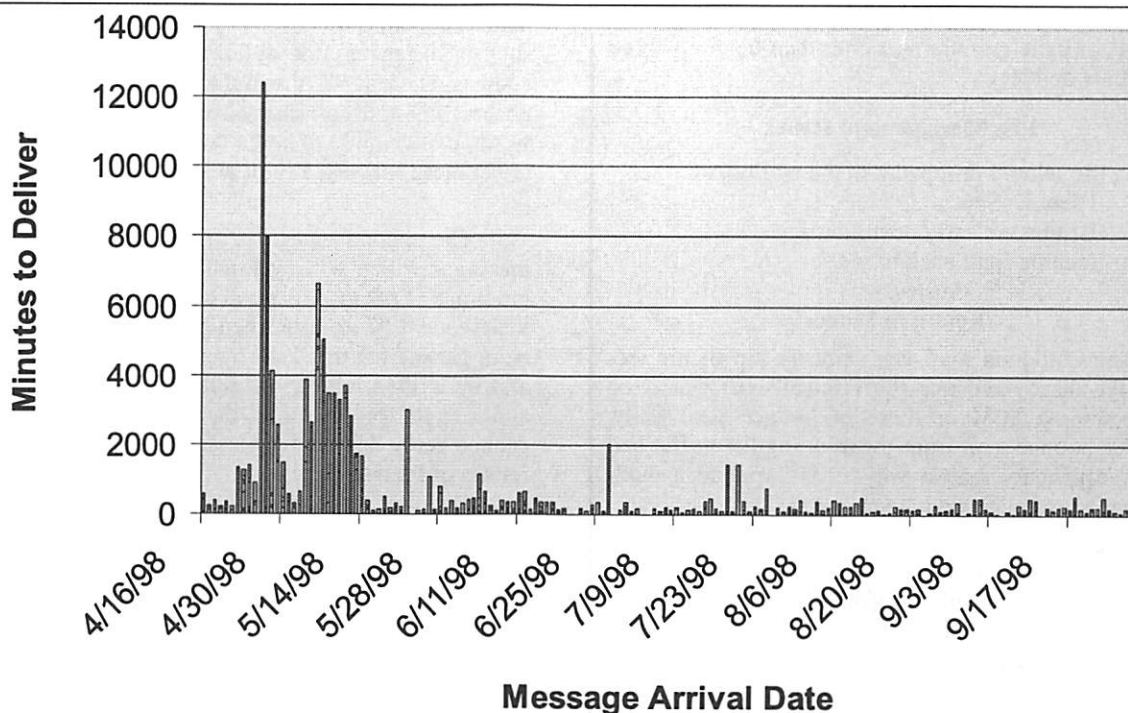


Figure 2: Message turnaround time.

Less than half a day after implementing this new queue processing method, only 7,000 recipients out of the initial 300,000 were still in the queue. All the recipients remaining in the queue were for "problem" addresses. At some point in the future, we may implement time-based queues as a subset of the spawn-managed queues.

Design Trade-offs

The astute reader will notice that it requires two passes of `qsplit` for a message to go from its initial bloated `qf` file in a holding directory to a chunked `qf` file in a live `sendmail` directory, awaiting delivery. This means that there is guaranteed to be at least five minutes, possibly as long as 10 minutes, between the generation of a list message by `Majordomo` and the earliest possible outbound opportunity for the message.

This is quite deliberate, for two reasons. The first, as you might guess, is simplicity of coding. Given that we were under a deadline to announce the list changeover, and that this level of rearchitecture had been slated for several weeks down the road, it was an expedient choice.

The second reason is directly functional for list purposes. Historically the Firewalls list has been plagued by flame wars of varying length and duration. We had been told that introducing a slight delay into message propagation, within reason, was the most expedient way to minimize the occurrence of flame-fests. Exchanging one-liners over a few hours rather than a few minutes tends to spoil a bit of the fun and allow cooler heads to prevail. Given the requirement for slightly delayed propagation, we chose to retain our 5-10 minute granularity rather than try for a more immediate delivery.

List Management Issues

"So when I dropped it in the mailbox,
I sent it 'Special D'
Bright and early next morning
it came right back to me."

— E. Presley,
"Return to Sender"

The outbound mail was only the tip of the iceberg. We discovered that the Firewalls list generated an astounding 80M or more of bounce mail daily. Keeping bounce mail from overrunning list traffic had been the primary reason why Brent went to a dual-sendmail system for the list years ago.

The `sendmail`-lists was set up for outbound mail only, with a conventional `sendmail` receiving incoming list traffic and the plethora of bounces. Brent and his compatriot, Michael C. Berch, had put together a series of scripts for winnowing the postmaster wheat from the bouncing chaff. While the scripts identified many user queries and passed them on for human action, the bulk of the mail was discarded. This meant

that secondary bounces would go uncorrected and trigger new recursive bounces.

Our plan from the beginning was to isolate bounce traffic even further, putting in a third separate `sendmail` structure solely for bounces. We would accomplish this by defining a virtual interface on the Internet-facing ethernet port and assigning it to "bounces.gnac.net". By tagging outgoing mail with `From` and `Reply-To` addresses at this host, we could control bounce traffic.

Automation scripts have been implemented in Perl, and have proven able to handle the formidable task of crunching through the huge volume of mail. We originally explored queuing the bounces via `procmail` as each message arrived, but quickly found that the overhead of calling `procmail` for each inbound message made batch processing of the bounce mail a better solution. The scripts, run out of `cron`, are explored below.

Automation of Bounce Handling

Since we had made the decision to automate wherever possible, we designed the script to identify and sort each message according to its potential for automation. Thus we arrived at three "bins" into which to toss processed bounce mail: "HUMAN," "AUTOMATABLE," and "JUNK."

The first category, JUNK, is for things which need throwing away. In particular, bounce messages to a bad address frequently generate second-generation bounce messages. We had planned on a scripting solution for these as well, but were pleased to note the "confDOUBLE_BOUNCE" option in the new `sendmail` 8.9 configuration. [15] Designed to catch just such occurrences, this option will let you specify an address, such as "`| /dev/null`," for these. To save on general I/O and processing wear and tear, however, it would be desirable to add a double-bounce ruleset and reject these messages right at the `check_compat` stage. [2]

The second category, AUTOMATABLE, is for messages which will eventually be handled by a programmatic response. A good example of this is the all-too-frequent "I unsubscribed but am still getting mail, help, get me off this list" query. A script will eventually be written which will pull out the sender's name and search for it in the database, then send off a canned reply describing message propagation and the results of the search.

Of particular interest in this category are routine bounce messages. We are in the process of adding a bounce manager which will extract addresses from a standard bounce message and process them. By hashing on the address and updating a counter, the script can quickly determine whether or not this is a repeat bounce offender. If so, the address can be automatically removed after a certain number of bounces. This represents a great improvement over the old

"bouncer" list functionality which required hand editing of the list files to accomplish.

The last category, HUMAN, is for what is left. These are items that usually require human intervention, either to answer the question posed, or to figure how this particular item can be automated successfully as above.

Spam

"There is one thing you must be sure of,
I can't take any more!"

— Peter Gabriel,
"Shock the Monkey"

Spam was a serious problem on the Firewalls list. The machine that the list runs from does not relay spam, but spam is sent directly to the list. When we took on the list, this was one of the issues we intended to tackle. Based on anecdotal information, we did not expect traditional solutions to scale to cope with the Firewalls list. As we looked into our options, an increasing quantity of Firewalls list traffic became discussions on how to make the list usable again from the perspective of the subscriber community.

One conventional approach to cutting down on spam is to block certain domains or IP address ranges from successfully sending mail through your sendmail daemon. This can either be done through sendmail configuration, or at the network layer using Vixie's black-hole BGP feed, or simple filters. These approaches would not work in our case because much of the Firewalls mail was still being forwarded from greatcircle.com, and that mail was a mixture of real messages and spam.

The simplest-sounding solution was to make the list a closed list, where only members can post. However, there were two problems with this. Firstly, a large number of the lists subscribers were local exploders at remote sites, whose membership we had no way of knowing. We did not want to prevent these people from posting, or to force them to all subscribe directly to the list. Secondly, Brent had concerns about how well the majordomo feature to do this would scale to a list this size, which is the reason that he had never activated it on his version of the list. He felt that the interlocking programs that make up Majordomo's interpreted Perl core could not feasibly keep up with the traffic.

Sendmail Database Approach

We also considered taking this same idea, restricting posting to list members, to a lower level, and having sendmail do the work via a database lookup mechanism. The members of both the Firewalls and the Firewalls-digest list would be automatically added to the database. In addition, a list called Firewalls-post could be created for offsite list exploder members who wish to post. The Firewalls-post list is maintained by majordomo so that subscribe and

unsubscribe requests can be handled automatically. All three lists would be made into a generic key/value sendmail database at regular intervals by a cron script.

We could trigger a database lookup only if the recipient was "firewalls" or "firewalls-digest." Otherwise we would end up screening out routine majordomo requests or postmaster mail. By positioning the lookup in the check_compat phase of mail processing, we would be able to reject unauthorized postings directly at the SMTP connect. Note that system addresses such as "firewalls-owner" and "majordomo" need to be included to allow normal majordomo operation. These must be qualified with the full host and domain name in order to prevent spoofing, e.g., "majordomo@lists.gnac.net" rather than just "majordomo."

Using Majordomo

Clearly the Firewalls-post list that was suggested for the sendmail solution to the "invisible subscribers" problem could also be applied to majordomo. Just to make sure that we weren't re-implementing the wheel for no reason, we also ran some tests to evaluate the overhead of using the majordomo "closed list" feature.

After running a set of tests using the majordomo restricted-posting lookups, we found that on our machine it took about three seconds to perform the lookup. We decided to accept this as part of the system overhead, and implemented this feature over the sendmail based one. We have considered implementing the sendmail based variant of this on general principle, however, and to evaluate its use as a solution for other large lists not using majordomo.

The final component is the communication piece. We forewarned the list membership that we were going to implement this feature, and gave the message a couple of days to reach everyone.

In addition, when majordomo rejects a message due to this feature, directions on subscribing to the Firewalls-post list are returned to the sender.

Potential Future Problems

Strictly speaking, a clever spammer could hand-set the sender to be a legitimate sender such as "majordomo@lists.gnac.net". It would be wise for us to include a "remote is identifying as me" ruleset as part of this, so that this kind of spoofing would be caught and detected with prejudice. [14]

If spammers monitor the list and start spamming under spoofed names of legitimate posters, we would have to up the ante and turn on the sendmail features which do host authentication via DNS. [1, 15] While this would impose more of a load on the server, our split sendmail configurations would allow us to implement this on the main inbound sendmail only, so that the performance hit would not be too severe. We hope to avoid this, as many legitimate sites have business reasons to aggregate traffic or architect their mail

infrastructure in such a way that they do not comply with strict sendmail checking.

Futures

"All the way to Malibu
from the Land of the Talking Drum:
Just look how far –
look how far we've come!"
– Don Henley,
"Building the Perfect Beast"

The cutover day for the Firewalls list move was April 15th, a red-letter day in its own right. Other than Brent's dual-sendmail structure, none of the facilities mentioned in this document existed on that date, nor had they been planned.

As of the writing of this paper, we are processing an average of 8184 messages per day. Turn around time for an individual message has dropped from pre-queue-split highs of 5-8 days to less than one day, and in many cases less than half an hour:

```
Statistics from Wed Apr 15 17:48:17 1998
M msgsfr byt_from msgsto bytes_to Mailer
0 0 OK 504929 2241822K prog
1 0 OK 3499 25520K *file*
3 402707 1831829K 270 914K local
4 51340 268777K 4584 59086K smtp
5 74148 421120K 2122 5626K esmtp
9 64 246K 0 OK uuwp-old
=====
T 528259 2521972K 515404 2332968K
date: Wed Jun 17 18:35:30 PDT 1998
```

In order to have better tracking of email flow through the list, we are intending to implement a script to take hourly snapshots of sendmail.st, process them, and feed the data to MRTG [16]. We have to do it that way since start/stop is impossible with so many send-mails all the time.

The script will need to aggregate the sendmail.st files of the variously spawned sendmails. They are separate files because sharing the same on the sendmail.st file.

When this is implemented, we intend to make the MRTG graphs available on the list website.

Further Processing

For lists with more "real-time" needs and less concern about flame wars, qsplit could be rewritten to deposit split files directly into processing queues at the time of splitting.

To improve message flow further, qsplit and spawner could be applied recursively to create time-based queues working with the existing spawn-managed queue directories. Messages over a certain age would be moved to a time-based queue and then split to a smaller \$CHUNKSIZE. By employing progressively smaller chunks, one could force the qf files down to one RCPT per message by the time they reached a particular age.

At this point, problem addresses would be identifiable automatically by their queue position. This could enable management of bad addresses completely outside of the traditional bounce/postmaster processing used by most list admins.

List Exploders

We'd like to eventually add some special-case handling for exploder lists. When we receive a generic individual user bounce via a remote exploder, it is very difficult to find the origin exploder and pass on the error to that list administrator. In fact, there is no distinction made in the Firewalls or Firewalls-Digest lists between individuals and exploders, so there are undoubtedly many exploders which are completely opaque to us.

One approach would be to increase dramatically the number of spawn-managed processing queues and set qsplit to always chunk RCPTs to one per message. We would further modify qsplit to add an RFC-822 compliant custom header [11] containing the envelope recipient to each qf file. This header line would be preserved in any remote mailer bounces, enabling us to see to which address the original message was sent. A bounce message whose "user not found" error did not match the address in the X-Custom-Recipient header could trigger a custom message to the address in the header, or be referred to a human administrator for hand-processing.

Availability

The scripts described in this paper will be made available at <http://www.lists.gnac.net> after the publication of this paper in December, 1998.

Conclusions

"Don't know much about history ..."
– Sam Cooke

Look before you leap!

Taking over the management of the Firewalls list seemed like an attractive proposition. It should be easy – just copy over Brent's setup on to faster equipment with better Internet connectivity and you're done. As we soon found out, it was not that simple.

Follow First Principles

There was no magic involved in turning the list into something that now runs smoothly. We stepped through a number of system administration basics. When the machine was in trouble, we looked at the hardware to see if more memory would help, and we looked at the kernel parameters and tuned them appropriately. After that, understanding the problems in detail and how the various solutions would affect the system allowed us to choose the correct course of action. Questioning our assumptions and gathering real data on which to base our decisions also proved worthwhile. Experimenting with different options off-

line to get real data without affecting the list is always the right approach for a production system.

Work smarter not harder

The machine was not CPU-bound, so throwing higher-end equipment at it would not help. It was also not even coming close to saturating our connectivity. What we needed to do was find a way to make the system work smarter. To that end, having understood the problems we looked at ways to split the recipient lists into smaller chunks, and at how to get multiple sendmail processes to constantly churn through the queues.

Communicate, communicate, communicate.

An important part of our work during the "hard times" when we had just taken over the list was to communicate with the readership and let everyone know what was going on, and that we were working on fixing each of the problems that arose. There were many people interested in helping out, and we got many interesting pointers from folks on the list (thanks folks!). Letting people know the list of problems that you are working on, and when you realistically expect to have them fixed is something we all need to remember to do. The implementor feels less pressured and the "customer" feels plugged in and listened to.

References

- [0] D. Brent Chapman, <http://www.greatcircle.com/lists/firewalls> (and posted to the Firewalls list).
- [1] "Majordomo: How I Manage 17 Mailing Lists Without Answering '-request' Mail," D. Brent Chapman, USENIX, *LISA VI Proceedings*, October 1992. ISBN 1-880446-47-2.
- [2] *Sendmail*, 2nd Ed., Brian Costales with Eric Allman, O'Reilly and Associates, 1997. ISBN 1-56592-222-0
- [3] *Mhonarc*, a Perl successor to mail2html, <http://www.oac.uci.edu/indiv/ehood/mhonarc.doc.html>.
- [4] Our "test mode" consisted of two parts. First, a parallel feed of Firewalls traffic provided by Brent. Second was merely the sending of a real message to the list recipients as part of a dry run. The message would be a precursor to the official announcements already drafted by Brent Chapman (Great Circle) and Christine Hogan (GNAC).
- [5] *Westworld*, Metro-Goldwyn-Mayer, [http://us.imdb.com/Title?Westworld+\(1973\)](http://us.imdb.com/Title?Westworld+(1973)).
- [6] *System Performance Tuning*, Mike Loukides, O'Reilly and Associates, 1992. ISBN 0-937175-60-9
- [7] *Sun Performance and Tuning*, Adrian Cockcroft, Sun Microsystems Inc., 1995. ISBN 0-13-149642-5
- [8] *Managing Internet Information Services*, Cricket Liu, Jerry Peek, Russ Jones, Bryan Buus and Adrian Nye, O'Reilly and Associates, 1994. ISBN 1-56592-062-7
- [9] Strata Rose, VirtualNet Consulting, Dave Ilstrup, WebAware; unpublished work 1995.
- [10] *Debian bulkmail*, <http://molec2.dfis.ull.es/debian/Packages/stable/mail/bulkmail.html>
- [11] *RFC-822: Standard for the Format of ARPA Internet Text Messages*, D. Crocker, August 13 1982.
- [12] "Tuning Sendmail for Large Mailing Lists," Rob Kolstad, USENIX, *LISA XI Proceedings*, October 1997. ISBN 1-880446-90-1.
- [13] *Multiple Independently Targetable Re-entry Vehicle*, http://www.janes.com/defence/resources/glossary/defres_glosmi-ml.html.
- [14] *Sendmail: Theory and Practice*, Frederick M. Avolio and Paul A. Vixie, Digital Press / Butterworth-Heinemann, 1995. ISBN 1-55558-127-7
- [15] Eric Allman and Sendmail Inc staff, <http://www.sendmail.org/> web site.
- [16] *MRTG, (Multi Router Traffic Grapher)*, Tobias Oetiker and David Rand, <http://ee-staff.ethz.ch/~oetiker/webtools/mrtg/mrtg.html>.

Request v3: A Modular, Extensible Task Tracking Tool

Joe Rhett – Navigist

ABSTRACT

Tracking tasks remains one of the most difficult issues facing any working team of administrators. Even with the implementation of commercial tools available today, e-mail and hallway conversations remain the standard for task management in many organizations; however, these make it difficult and time consuming to remain current on issues, and do nothing to summarize the long-term history of tasks and completion thereof.

Many commercial tools are available to handle task management, and most work quite well for stereotype models of their intended environments – development teams, help desk, etc. Unfortunately, these systems often have limitations which prevent their use (or a simple deployment) in a pre-existing, working environment. Other systems are difficult or time-consuming to use, and remain ignored in favor of task accomplishment. Few freely available systems provide the statistics to analyze productivity, generate statistics, and otherwise please management.

Request v3 was designed to provide the necessary essentials for modern task management: a selection of user interfaces, support for multiple database backends, flexible security controls, and extensive reporting capabilities. It runs cleanly in heterogeneous environments, including those that have a large installed base of Windows users. It includes command line, e-mail, and web interfaces, in addition to an Extension Interface which provides a simple way to access the Request system from other programs, scripts, or any custom interface one may create. The authentication, notification, data storage, and logging functions are processed within separate modules, allowing a variety of backend databases to be supported.

Introduction

This paper focuses on the modifications and extension of the venerable task tracking tool Request, and how it may be effectively utilized to increase the response capability of any organization that does not have an effective, well-used task management tool.

The paper starts with an overview of the motivation behind the project and some history of Request itself. The following sections present the changes made and the functionality added to the system. The final section presents potential and operational uses for Request v3.

Motivation

Although our team of administrators were quite effective at using e-mail for task management, we believed that not enough information was available to our users about task status, and changing responsibility for the task was never handled well. Although our staff was competent, we could not step in to handle emergencies in another's field of responsibility. No information was available to analyze our performance, or set expectations for our users.

The initial goal was to find a simple system we could quickly implement to track tasks. Given the extensive automation in our environment and the highly mobile nature of our user base, it was clear that

any solution for task management must support the existing environment, work with the existing systems, and make it easier for administrators to perform their jobs, not add to their existing burden. The solution had to provide additional functionality without requiring changes to our software or current administration style. The goal became one to find a system which would provide this baseline functionality for task management in an operations environment.

After analyzing the existing commercial solutions, it was clear that none of the solutions directly supported our existing environment of mixed version Intel, Sparc, and HP unices with a growing amount of Windows NT systems. None of them would support all of the methods of input – command line, web-based, e-mail, and non-interactive – necessary. Most of the solutions required extensive investments in time and money (Remedy, Clarify) to be fully operational, yet would not integrate seamlessly into our environment. There simply was not a practical solution we could quickly implement.

At this point, we began to examine the freely available products. As with the commercial products, these packages worked seamlessly for one or two environments, but did not handle everything we were required to support. Many of the latest ones did not include command line or non-interactive input (Jitterbug, PTS, troublemh), and some of the web interfaces

were obvious hacks (GNATS). None of them included clean support for MIME e-mail. None of them provided even a basic, functional set of statistics that our management desired.

The available products fell into two categories: simple, useful systems for tracking progress, and complex systems that provided statistics. None of the systems had both in good measure, and all of them required further change to make a viable system capable of supporting our diverse systems and Windows-based user community. We needed something we could quickly add the missing functionality to, and start using immediately. In the end, this was perhaps the strongest criteria.

We chose Request 2.1c as our starting point due to the fact that I was able to install and use it within an hour, and add the essential missing functionality within an afternoon.

History

Request was originally written by Shawn Instenes and James Sharp of Lawrence Livermore National Laboratories. It started as a single, large script written to manage the tasks of a specific set of administrators assisting Unix workstation users. This KISS¹-style system works well in an environment of trusted users, each having accounts on the system where the database was stored.

Request 2.1 provided the following features:

- Command line and interactive interfaces
- The ability to open, close, assign, and update tasks
- The ability to set assigned and default due dates for tasks
- A defined list of who the available assignees where
- The ability to list open and closed tasks within the last week or month

Request 2.1c (UMI): Stuart Levy (University of Minnesota) added support for e-mail and a proto-standard query interface, which allowed reports to be generated using keywords and values. Unfortunately, these routines contained their own implementations of each function and had little data verification.

Request 2.5 (LLNL): Mike Miller had updated the original source tree, and began to generalize the routines and remove some redundancy. He also added printer support and defaults to the prompts.

Request 2.6-alpha (LLNL): Mike Miller added NFS-friendly locking code, and time spent and priority fields to the record format. The time spent and assigned fields became arrays, providing a history mechanism.

¹KISS: Keep It Simple, Stupid – a philosophy of programming which accents functionality rather than the addition of extra features.

When I first started working with the code, my immediate goal was to provide functional e-mail and web interfaces. Stuart Levy's e-mail interface was functional, but performed all of its own data processing, which sometimes made it incompatible with the command line interface. In particular, dates were input in different ways, making the limited reporting functionality almost useless.

Due to the lack of standardized methods in the code, I was not able to easily add a web interface. To fulfill the immediate need, I created a quick web interface by checking the data input and using the command line features of Request. The data was provided back in fixed-width, preformatted output. This was not the best approach, but fulfilled the immediate need.

Even after all of these changes, Request remained lacking in the following ways:

- No security (all files mode 777)
- A web interface was not provided
- MIME e-mail was not supported
- Code was inconsistent, sometimes non-functional
- Comprehensive statistics were not available
- The ability to notify or alert about status was not available

Design Goals

Review of the system revealed many problems within the code, mostly related to improper assignments, and logging changes before testing whether to perform said changes. The code base was inconsistent and redundant, where each function reimplemented every operation, often in incompatible or inconsistent ways. Adding new features to the system required modifications to every function. The "simple task" of integration became a complete re-write of the system.

Four major issues define changes to the code base: ease of use, remote access, security, and interoperability. The system is designed to:

1. Secure the dataset against unauthorized operations or direct attacks.
2. Support local and remote users on Unix, Windows, and Mac platforms.
3. Provide an intuitive system which does not require instruction to use.
4. Process non-interactive input and output from existing automated systems (for example, accept input from HP OpenView, and make use of our alpha-paging software).

To prevent similar problems in the future, the new design centralizes all data access, verification, and logging. In addition to resolving the inconsistency and redundancy issues, these changes make it simple to add a modular Extension Interface, allowing new methods of access to the task information with a minor amount of code.

Major Feature Changes

1. Year 2000 Compliant – code automatically fixes millenia problems when loading/saving tasks
2. MIME e-mail capability – can parse and add plaintext sections of MIME messages.
3. E-mail, web, and interactive interface use the same common routines.
4. Standardized logging mechanism.
5. Common notification methods.
6. Configuration option for European date format.
7. Pointers to data structures are used, rather than copying arrays between each routine.
8. Clean code interface allows simple extensions without breaking upgrade capability.
9. New fields store information regarding time spent and task priority, and retain a history of assignments for each user.
10. Debugging can be enabled at any time within the program (limited by authorization).

Common Routines

As mentioned before, the biggest problem in the old Request code was the lack of centralized data control. An environment which forces every function to reimplement data validation and storage will develop inconsistencies in the different implementations. Request 2.1c is perhaps the worst example of this, where dates were stored in a different format (Feb25 vs 2/25) by the e-mail interface, making the date-oriented reporting absolutely useless.

Request v3 uses a common set of routines for all data access, providing consistent data validation and verification, thus relieving the burden from the interface. A reference to the data is returned from the method, allowing the interfaces to input and output in any fashion desired. It becomes trivial to design interfaces for non-interactive scripts.

The centralized system makes it easy to add external functionality. For example, the Notify module can interact with other management applications, such as management software, alpha-pagers, and pop-up windows. The documented API used by the modules

allows these additions without modification to the source code, thus allowing live testing and integration.

Interfaces

Command Line

The command line interface is backwards compatible with the previous versions of Request, but adds additional functionality not present in the previous versions:

1. Configurable default action (default was to create a new request, is now often interactive)
2. Chaining of commands (“request -u 1234 -c -Q 1234”)
3. Word forms for all the actions (“update 1234”)
4. Extended search criteria (“find state open,assigned jrhett,contains Print”)²
5. The ability to enable debug output within the program
6. The ability to create new requests while in interactive mode
7. Command history mechanism ala c-shell
8. Optional direct access to routines for testing purposes (limited by authorization)²

E-Mail

The e-mail interface, originally written by Stuart Levy, performed all data operations itself and was therefore sometimes incompatible with the main routines. The e-mail routines were rewritten to utilize common data access routines, and handle messages in MIME format.

Experience and user input lead to the implementation of these features:

- External files for custom response messages
- Immediate feedback to the user³
- Automatic processing of looped or failed messages

²These features were thought of and partially implemented by Stuart Levy. We simply extended the ideas a bit further, or rewrote it completely to provide the intended functionality.

³Actually implemented, then removed by Stuart Levy when he decided it was too annoying; obviously optional.

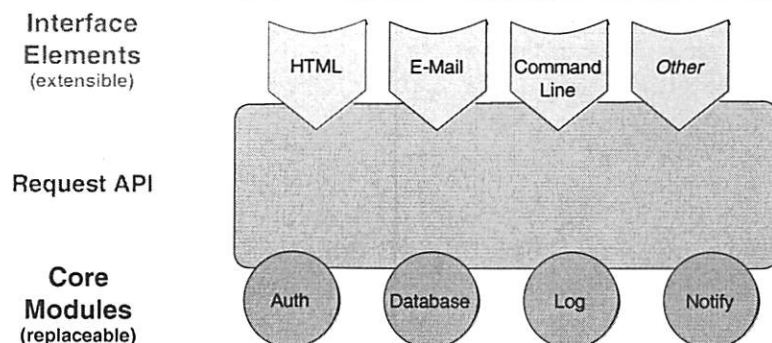


Figure 1: Request structure.

One site of which I am aware has added encryption and digital signature authentication to the e-mail interface.

Web Interface

Supporting a wide variety of users requires the ability to provide a variety of interfaces, accessible in a method understood by each user community. Users do not always take the time to learn an interface, especially when they already feel inconvenienced by some issue; therefore, we found it important to provide an interface with which most users feel comfortable. Over the last few years it has become apparent that the web browser is the most common, usable interface.

The web interface uses the CGI environment to receive data from HTML forms and return output in HTML style. The default output is an HTML table not dissimilar from the command line output, but can be easily modified to output in any format desired. It seems likely that every site will use a different style, so customization is left to the site administrators.

This is actually an example implementation of the Extension Interface.

Extension Interface

The Extension Interface provides the ability to support new interfaces and custom access methods without modifications to the original code base. Originally designed to interact with HP OpenView NNM and PPT's E-Page software, the interface easily adapts to most needs. Perl/TK and other interfaces would be simple implementations for enhanced user input mechanisms.

The interface is well documented. Skeleton and example invocations are provided to assist new development.

Authentication, Storage, Notification and Logging

All of the following components are PERL classes (Request::Authentication, Request::Storage, etc) implementing a documented interface. Any or all of these could be replaced by custom modules utilizing different resources, such as an existing Oracle database or an LDAP server. An example module of each type is supplied to assist in testing.

Authentication Module

Request v2.1 performed no authentication, allowing any user to execute any command. While this obviously is not sufficient for complex environments, anything which enforces stricter security may complicate administration unnecessarily. Therefore, version 3's authentication module provides flexible security controls, allowing everything from relaxed, open administration to strict, multilevel authentication which validates each operation against the user's rights.

The authentication information can be accessed from any source. The default module uses the local system's user information, but modules which support DBM databases, Radius, LDAP, or any other system could be easily created.

Database Module

By moving all data storage and retrieval operations into a module, data can now be stored in one of any number of different backend databases. This provides the ability to manage data using a site's current environment. Standardized tools can be used to manage, distribute, and generate reports from the data, instead of Request-specific code.

The standard DBM database provides the default database support, to retain backwards compatibility with older versions.

Logging Module

Unlike Request v2, extensive support for basic, extended and debug logging is available. Similar to the other modules, the logging routines can be replaced at will.

The default logging module provides direct file logging with a priority control. An additional module supplies standard syslog() logging methods.

Notification Module

Unlike Request v2, extensive support for basic, extended and debug logging is available. Similar to the other modules, the logging routines can be replaced at will.

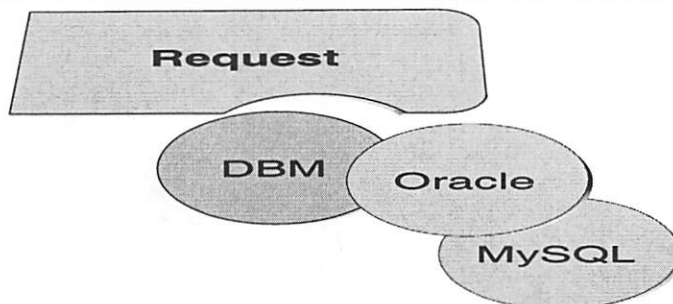


Figure 2: Request module.

The default logging module provides direct file logging with a priority control. An additional module supplies standard syslog() logging methods.

Reporting and Statistics

One of the significant differences between the freely available systems and the commercial systems is that the commercial systems have extensive capability to produce reports and statistics of any flavor. The freely available systems usually provide somewhat limited functionality, often being concerned with ease of use and functionality. Managers often need reports and statistics to justify budgets and review resource allocations. The implementation of commercial systems is often based upon that criteria. Far too often administrators are saddled with unfriendly, cumbersome task management tools, simply because their managers need more reporting capability.

Managers desire methods to review activity, the project history, and statistics on task completion and overall performance. And sometimes the pointy-haired ones just like pretty pictures. In the trenches, administrators need to justify their time and resource allocations to their management, and identify problem areas which strain the group as a whole.

Request v2.x provided only limited support for reports, mostly oriented towards those which were based upon the open and closure of tasks. Request 2.1c provided a basic query language for generating reports based on a variety of criteria. Testing found that not all of the functionality was implemented, and many of the reports were not useful due to a lack of data validation in the input routines.

The Request v3 reporting engine extends the query language defined by Stuart Levy, providing access query functionality with a standard format. Stronger, common validation provides more accurate reporting. Reports can be generated against any field, including custom fields defined in the local configuration. Thus, changes can be made to the data structure without modification to the reporting engine.

An optional extension of the reporting engine will create GIF graphs from any single or dual-column numeric report.

Security

The previous versions of Request provided little security. All of the shared data areas were world-writable, and changing or removing information from the database was trivial. Request v3 removes all need for globally accessible directories, implementing access using a Unix group membership.

This is considered somewhat basic. The best security comes from using the modular structure to store data in a backend database (Oracle, MySQL, etc), which implements a much stronger security mechanism.

Internally, Request v3 includes a per-operation authorization mechanism. By default it remains as open as the previous versions, but may be configured as tight as the local practice requires. This is discussed in the Authentication Module section.

Implementation

Installing the Package

If you have fought your way through installation of many older task management tools, you will be pleased with the installation process in Request v3. Unlike the previous versions of Request, installation is simply make install. The installation will ask a few questions, and installs a working system. After installation, the configuration file may be edited to provide additional customization, but this is often unnecessary for a basic installation.

The installation program attempts to find an existing Request installation, and prompts for action if found. If you choose to upgrade, the existing database will be available from the new version. If you choose to duplicate the existing configuration, both systems may be used during a test period. In either case, the installation program will automatically configure Request v3 to match your old configuration, allowing immediate use of the new system.

```

# A minimal example which changes the assigned admin
use Request;

# Get the command line input
$person = $ARGV[0];
($day,$month,$year) = (localtime(time))[3,4,5];
$tomorrow = ($month + 1) . "/" . ($day + 1) . "/" . ($year + 1900);

# Open the request and reassign it to the person
my $request = $Request->retrieve($id);
$request->assign($person);
$request->changedue($tomorrow);

```

Listing 1: Adding a new interface.

Customizing the Features

All of the standard configuration parameters are contained within a single text file. Unlike previous versions of Request, this file is not a PERL script, so values do not need to be quoted or escaped!

```
DefaultAction -i
DefaultDue    2days
MailCommand   /usr/lib/sendmail -t
```

It would be very simple to implement an integrated configuration editor, but it has not been necessary.

Adding New Interfaces

A new or improved interface may be installed and tested without affecting normal operation. The interface needs to do nothing more than use the PERL class, create an object (if necessary) and call the methods; see Listing 1.

To assist in development, we have supplied a dummy interface which utilizes every available method of the Extension interface, but contains only comments for input/output code.

Adding New Modules

Any of the Authentication, Storage, Notification, or Logging modules may be replaced at any time. These modules are stored in the /lib path of the installation directory. A replacement module may be tested by changing an environment variable, allowing testing without interrupting normal operation; see Listing 2.

To assist in development, we have supplied a complete example for each module which implements each function of the API, but does not actually do anything. These examples provide skeletons from which to begin development.

Availability

Request v3.0 is publicly available using anonymous HTTP (web browser) at <http://www.navigist.com/Reference/Projects/Request>.

A mailing list has been created for questions, comments, patches, and recommendations for Request. You can subscribe by sending electronic mail to majordomo@lists.isite.net with the text "subscribe request-users" in the body.

Author Information

Joe Rhett started his career as an independent contractor, implementing Unix systems and LANs in the Washington, D.C. area. After working with the NAVSEA MAN in Crystal City, VA; he moved to California, and began work with Navigist, a small team of consultants in Silicon Valley. He specializes in network engineering, security, and performance; often focusing on application implementation. He tries to limit his programming to small applications and utilities which improve administration capability, problem isolation, and overall response time. He can be reached via e-mail at [<jrhett@navigist.com>](mailto:jrhett@navigist.com), or through any of the contact methods listed at <http://www.navigist.com/Staff/JRhett>.

References

- James M. Sharp, "Request: A Tool for Training New Sys Admins and Managing Old Ones," *Proceedings of the Sixth USENIX Systems Administration Conference (LISA VI)*, 1992.
- Stuart Levy. "README for Request v2.1c," *Request 2.1c distribution*, July 1996.
- Mike Miller. "README for Request v2.5," *Request 2.5 distribution*, April 1995.
- N. Freed & N. Borenstein. "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies," *RFC 2045, Network Information Center*, 1996.
- N. Freed & N. Borenstein. "MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies," *RFC 1521, Network Information Center*, 1993.

```
# A minimal example
package Request::Log;

# logfile opened in init() and closed in final()
sub log {
    my ($pkg,$routine,$user,$level,$message) = @_;
    print LOGFILE "${level}/${pkg}:${routine}/${user}: ${message}\n"
}

# This assumes you only use debug in command line mode
sub debug {
    my ($pkg,$routine,$user,$level,$message) = @_;
    print STDERR "${level}/${pkg}:${routine}/${user}: ${message}\n"
}
```

Listing 2: Adding a new module.

- N. Freed & N. Borenstein. "MIME (Multipurpose Internet Mail Extensions): Mechanisms for Specifying and Describing the Format of Internet Message Bodies," *RFC 1341, Network Information Center*, 1992.
- D. Crocker. "Standard for the format of ARPA Internet text messages," *RFC 822, Network Information Center*, 1982.
- Sriram Srinivasan. *Advanced PERL Programming*, O'Reilly and Associates, 1997.
- Larry Wall, Tom Christiansen, Randal Schwartz. *PERL Programming*, O'Reilly and Associates, 1996.

USENIX ASSOCIATION

USENIX is the Advanced Computing Systems Association. Since 1975, it has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world.

USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems.

The USENIX Association and its members are dedicated to:

- problem-solving with a practical bias,
- fostering innovation and research that works,
- rapidly communicating results of research and innovation,

- providing a neutral forum for the exercise of critical thought and the airing of technical issues.

USENIX holds an annual multi-topic technical conference, the annual Systems Administration (LISA) conference, and frequent single-topic symposia addressing topics such as UNIX security, Tcl/Tk, object-oriented technologies, networking, electronic commerce, and operating systems design – as many as ten technical meetings every year. It publishes a magazine, *login*, eight times a year; and proceedings for each of its conferences and symposia. It also sponsors special technical groups as well as participating in various standards efforts such as IEEE, ANSI, and ISO.

SAGE, the System Administrators Guild

SAGE, a Special Technical Group within the USENIX Association, is dedicated to the recognition and advancement of system administration as a profession. To join SAGE, you must be a member of USENIX.

SAGE activities currently include publishing the "Short Topics in System Administration" series, the first three of which are "Job Descriptions for System Administrators," "A Guide to Developing Computing Policy Documents" and "Systems Security: A Management Perspective"; "SAGE News & Features," a regular section in *login*; The System Administrator Profile, an annual survey of system administrator salaries and responsibilities; co-sponsoring the LISA, SANS, and Large Installation System Administration of Windows NT conferences; support of working groups; encouraging the formation of local SAGE groups; and an archive site for papers from the LISA conferences and sysadmin-related documentation.

As a member of the USENIX Association/SAGE, you receive:

- Access to the papers from the USENIX Conference and Symposia proceedings, starting with 1993, via the USENIX Online Library on the World Wide Web (this includes the 1993, 1994, 1995, and 1996 LISA Conferences).
- Free subscription to *login*, the Association's magazine, published eight times a year, featuring technical articles, SAGE News & Features, columns on tools and techniques,

book and software reviews, summaries of sessions at USENIX conferences, snitch Reports from various ANSI, IEEE, and ISO standards efforts, and much more.

- Discounts on registration for technical sessions at all USENIX conferences and symposia.
- Discounts on the purchase of proceedings and CD-ROMs from USENIX conferences and symposia.
- PGP Key signing service (available at conferences)
- Discount on the 4.4BSD Manuals plus CD-ROM published by the USENIX Association and O'Reilly & Associates, Inc.
- Special subscription rate to The Linux Journal.
- 20% discount of all titles from O'Reilly & Associates and Prentice Hall PTR.
- Savings (10-20%) on selected titles from McGraw-Hill, The MIT Press, Morgan Kaufmann Publishers, Sage Science Press, and John Wiley & Sons.
- Discount on all publications and software from Prime Time Freeware.
- Discount on software from BSDI, Inc.
- Right to vote on matters affecting the Association, its by-laws, election of its directors and officers.
- Right to join Special Technical Groups such as SAGE.
- SAGE members receive the following additional benefits: The most recent pamphlet in the "Short Topics in System Administration" series and the annual System Administrator Profile.

Supporting Members of the USENIX Association:

ANDATACO
Apunix Computer Services
Auspex Systems, Inc.
Cirrus Technologies
Cisco Systems Inc.
Compaq Computer Corporation
CyberSource Corporation

Earthlink Network, Inc.
Hewlett-Packard India Software Operations
Internet Security Systems, Inc.
Invincible Technologies Corporation
Lucent Technologies, Bell Labs
Microsoft Research
NeoSoft, Inc.

Nimrod AS
Performance Computing
Sun Microsystems, Inc.
TeamQuest Corporation
UUNET Technologies, Inc.
Windows NT Systems Magazine
WITSEC, Inc.

SAGE Supporting Members:

Atlantic Systems Group
Collective Technologies
Compaq Computer Corporation
D.E.Shaw & Co.
ESM Services, Inc.

Global Networking & Computing Inc.
Microsoft Research
O'Reilly & Associates
Remedy Corporation

SysAdmin Magazine
Taos Mountain
TransQuest Technologies, Inc.
Unix Guru Universe (UGU)

For more information about USENIX and SAGE membership, conferences, or publications, please contact:

The USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710

Phone: +1 510 528-8649
Fax: +1 510 548-5738
Email: office@usenix.org
WWW: <http://www.usenix.org>

